Feasible Form Parameter Design of Complex Ship Hull Form Geometry

Thomas L. McCulloch
University of New Orleans, tmccullo@uno.edu

Follow this and additional works at: https://scholarworks.uno.edu/td

Part of the Artificial Intelligence and Robotics Commons, Computer-Aided Engineering and Design Commons, Graphics and Human Computer Interfaces Commons, Numerical Analysis and Scientific Computing Commons, Ocean Engineering Commons, and the Software Engineering Commons

Recommended Citation
https://scholarworks.uno.edu/td/2552

This Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UNO. It has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. The author is solely responsible for ensuring compliance with copyright. For more information, please contact scholarworks@uno.edu.
Feasible Form Parameter Design of Complex Ship Hull Form Geometry

A Dissertation

Submitted to the Graduate Faculty of the University of New Orleans in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering and Applied Science Naval Architecture & Marine Engineering

by

Thomas Luke McCulloch
B.S. Auburn University, 2004
MBA Auburn University, 2007
M.S. University of New Orleans, 2010
December, 2018
To Erin.
The privilege of writing a dissertation would not be possible without the help of so many people.

First and foremost, the advisors. Lothar, thank you for guiding me along this journey by suggesting some excellent problems and guiding me back where I might have sailed too far off the path. Baker, I really lucked out to find you down the road at LSU Lafayette. You knew exactly what I needed to hear and at just the right time. Dr. Vorus, thanks for giving me a chance to “work on the hard problems” way back in the beginning of my time in engineering grad-school.

Will Byrd, thank you for giving me some of your time to talk intervals in and around miniKanren. Thanks, Chris McKesson, for being so welcoming after I had been away. Thanks other Baker, you showed me how its done, and Chase for all the fun discussions in the grad office. Thanks Ryan, for years of good jokes and lots of fun messing with gear in the wave tank. We still need to attempt a jam session. Maybe I’ve got time now. Thanks George, for years of bad jokes and lots of fun messing with stuff in the wave tank too.

Thank you everyone at NOLA Hack Night and the NOFUN’ctors of NOLA. (Thanks to you guys for letting me get up and talk about engineering and... Python! You always made me feel welcome.) Thanks especially to Hunter for introducing me around, and letting me drink beers in your sweet floating geodesic dome during bayou boogaloo. If anybody actually reads this, you could do worse than moving to New Orleans for your PhD.

Thanks very much, Bentley Systems, for giving me some free time to work on this. Thanks particularly to the guys in the SACS office, for putting up with my comings and goings, and to Phil Christensen, for taking me on in the first place.

Finally, I have so much thanks to give to my family, who have always been there for me. Thanks especially to my wife Erin, who’s unwavering support has carried the day. Finally, thank you Padfoot; without your appearance I might never have finished this thing!
## Contents

List of Figures ......................................................... vii
List of Tables ......................................................... xi
List of Listings ......................................................... xii
Abstract ............................................................... xv

1 Introduction ......................................................... 1

2 Related Work ......................................................... 11

3 Form Parameter Design of Ship Hull Geometry ................. 36
   3.1 Introduction and Greater Context .......................... 36
   3.2 Procedures .................................................. 38
   3.3 Hull Form Generation ...................................... 39
   3.4 Curves of Form ............................................ 40
   3.5 Transverse Curves ........................................ 49
   3.6 Longitudinal Curves ....................................... 51
   3.7 Summary ................................................... 54

4 Lagrangian Curve Design of B-splines ......................... 55
   4.1 Constraints ................................................ 57
   4.2 Objective Function ....................................... 60

5 Automatic Differentiation Scheme ............................... 65
   5.1 B-spline Evaluation Example ............................. 70

6 Automatic Differentiation of B-splines ......................... 75

7 B-spline Curve Design Examples ................................ 78
   7.1 SAC Design and Optimization ............................ 79
   7.2 Design of a Stern Profile Curve ......................... 82
   7.3 Finding a Fair Enclosure ................................ 85
   7.4 Summary ................................................... 87

8 Picking Feasible Form Parameters ............................... 88
   8.1 Interval Analysis .......................................... 90
   8.2 Constraint Logic Programming Scheme ................... 117
   8.3 Bringing Intervals together with Constraint Programming ... 128
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>An Internal Domain Specific Language for Constraint Propagation</td>
<td>155</td>
</tr>
<tr>
<td></td>
<td>9.1 Implementation: Compiling Logical Rules</td>
<td>157</td>
</tr>
<tr>
<td>10</td>
<td>Truncated Hierarchical B-splines</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>10.1 Introduction to Multiresolution Modeling</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>10.2 Choosing a Representation And THB-spline Theory and Implementation</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>10.3 Important Properties Satisfied by THB-splines</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>10.4 Multilevel Editing</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>10.5 Multilevel Optimization</td>
<td>187</td>
</tr>
<tr>
<td>11</td>
<td>Final Program</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>11.1 Random Form Parameter Input Generation</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td>11.2 Complex Hull Form With Bulbous Bow</td>
<td>201</td>
</tr>
<tr>
<td></td>
<td>11.3 Usage of THB splines to Assemble the Complex Hull</td>
<td>206</td>
</tr>
<tr>
<td>12</td>
<td>Results</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>12.1 Plots of Generated OSV Hull Geometry</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>12.2 Validation by Tables of Results</td>
<td>246</td>
</tr>
<tr>
<td>13</td>
<td>Summary and Conclusions</td>
<td>253</td>
</tr>
<tr>
<td></td>
<td>13.1 Future Work and Outlook</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td><strong>Bibliography</strong></td>
<td>261</td>
</tr>
<tr>
<td></td>
<td><strong>Vita</strong></td>
<td>276</td>
</tr>
</tbody>
</table>
### List of Figures

1.1 From Abt and Harries, [1], Cost vs. Time, top, and Knowledge vs. Time, bottom. This represents the knowledge gap. Expenses fixed by design decisions become large shortly after a project commences, but the knowledge accumulated is at a minimum here. Therefore, we strive to increase the amount of knowledge early on in the design phase, and so close the gap.  

3.1 Bare hull form generation process workflow for feasible form parameter design  

3.2 FPD Process Example: Using SAC to specify area at hull stations. Figure from Birk, [2].  

3.3 Normalized Sectional Area Curve Showing flat mid-section and fore and aft transverse area accommodations for the eventual bulbous bow and transom stern. Longitudinal center of gravity is also fixed in this curve, though not shown here. See section 7.1 for more details on SAC optimization. An overview of the form parameters used for this particular curve, and thus those used in the eventual hull forms developed here, is given below in table 3.2.  

3.4 Normalized Sectional Area Curve Also showing concurrently designed Bulbous Bow Sectional Area Curve and fairing area (small curve to bottom left, where the flat portion is actual bulbous bow and the angled portion handles fairing from the bulbous bow to the hull proper). Furthermore, the bulbous bow itself is the flat section of the smaller curve, while the inclined section shows that the fairing of the bulb into the hull will show an increasing sectional area moving aft. Note that fairing curve can be influenced by rule. This will be important later in the thesis for “nicely” integrating the complex hull.  

3.5 Normalized Designed Waterline Curve.  

3.6 Normalized Center-plane Keel Profile Curve. Forward sweep developed (at bow, left) in anticipation of the addition of the bulbous bow. Transom “drop” aft is controlled by $y_E$.  

3.7 Normalized primary SAC, bulbous bow and blending section SAC, and a designed waterline and center plane keel profile curve. The bulbous bow SAC is the short curve visible at bottom left. The fairing area is shown as well, though hard to discern here. The full length curves at bottom are the design waterline and center plane keel. Note that the design waterline (half breadth) and center plane keel have similar magnitude of maximum depth and magnitude of maximum width on this vessel. We might use a draft to beam rule if we wish to control this.  

3.8 Simple transverse curves network. This view shows the longitudinal positioning of these curves. Bow curve at far right. Transom curve at far left. An initial bow curve is shown. It will be replaced when the form parameter designed bulbous bow is added.  

3.9 Transverse section curves.  

3.10 Simple (bare hull) Longitudinal Curves. This is what a set of bare hull curves would look like without modifications to support bulbous bow incorporation.  

3.11 Simple (bare hull) Longitudinal Curves. Note that the form parameter designed bulbous bow has not been added at this stage, but the center-plane keel profile curve has been swept forward in anticipation of the new bulb. Future work could ensure that center plane area and distribution of area remained constant when comparing bare hull to complex hull.
4.1 Desired curve properties defined by a selection of form parameters ........................................ 56
7.1 Initial SAC curve ................................................................. 79
7.2 SAC curve development ......................................................... 80
7.3 Run Profile (This is a single B-spline curve). Note the relative constraints, \( r_1, r_2, \) and \( r_3, \) as well as the particular cusps, \( k_1 \) and \( k_2, \) which enforce user chosen geometry while allowing the local assembly to remain free to move relative to the rest of the control points. ........................................ 83
7.4 Curve enclosing a box ............................................................. 85
8.1 Annotated computational syntax tree for the constraint: \( (x_1 + x_2) \times x_3 \in [1, 2]. \) In our program, we will use Python’s native parsing plus operator overloading to produce such trees automatically from mathematical input. ................................................................. 102
8.2 Creation of intermediate terms, \( a \) and \( b, \) which will allow interval values to propagate up the syntax tree. ................................................................. 103
8.3 Forward propagation. The algorithm carries out the basic computations of the rule, contracting the interval widths where possible. ................................................................. 104
8.4 Annotation tree for the backward propagation of the constraint. Dummy variables again aid in propagating data — this time back down the tree to the terminal interval nodes, \( x_1, x_2, x_3. \) ................................................................. 105
9.1 Example Computational Graph Using Naval Architecture Rules. Ellipses represent design variables which have not yet been used by the designer as explicit rules. Boxes represent those form parameters which have. (Any mathematically defined term is also a rule. Computationally, if only for graphing purposes, we distinguish here between those that have appeared on the left hand side of our relational equality “double equals sign” and those that have not yet done so.) Lines represent paths of mutual dependence between form parameters. Self intersecting lines represent relations that also define design variables. ................................................................. 163
9.2 Code transformation procedure enacted by the \texttt{PStates} class under composition to form compound mathematical relationships and transform them into relational, logical rules. ................................................................. 170
10.1 Multilevel THB refined Bare Hull before deformation. Transverse curves shown in orange. (Displayed in Python’s Matplotlib library because \texttt{Rhino} does not support hierarchical splines, but only allows T-splines which we excluded from consideration in section 10.2 ) ................................................................. 185
10.2 Multilevel THB Hull after Bulbous Bow deformation with THB-spline refined control vertex net. Higher vertex counts denoted by blue circles denoting vertices at one higher level of refinement as compared to the magenta vertices aft. Please note that this single surface THB-spline will be refined further, especially the forward surface portions, and then projected to a B-spline representation at the highest level of detail and sent to Rhino for final display. See below for details and example hulls which have been further processed in this way. ................................................................. 187
10.3 Adaptively refined and solved longitudinal curves. Here, portions of the longitudinal space curves forward of midship have been refined. ................................................................. 188
10.4 THB-spline curve optimization algorithm flowchart ................................................................. 191
11.1 Final output of the form parameter process: A multi-surface complex hull. ................................................................. 195
11.2 Basic algorithm using arc consistency for random, yet consistent, form parameter design selection. Note that during numerical experiments, the design space and rules could typically be searched over without very much, or any, backtracking needed. ................................................................. 198
11.3 Complex Hull Form Generation: Overview of the Process ................................................................. 201
Forward Fairness Curve: shown here, in gold (or lighter color if viewing in black and white). The relative longitudinal position of this curve is a “hyperparameter” of the program, as are the relative forward position (from bow to midships sections) of the other forward volume defining curve and the relative aft positions (from midships sections to stern) of the aft volume defining curves. The area of this particular curve, as with any transverse station curve, is derived from SAC. The following information sets this curve apart from other fore and aft volume curves: The fraction of this curve’s area which is contained in the lower “bulbous” or here, simply rounded, portion of this curve is another input to the system which can be chosen by the user. Since it will be known in advance, it is used to set the height of the bulbous bow fairing sectional area curve at this station. This is an additional curve which shows the volumetric distribution utilized in joining the bulbous bow to the bare hull. This curve is of great utility in the design and integration of a bulbous bow, as discussed in section 11.2.

Procedure to fit the bare hull to the bulbous bow in such a way as to match naturally the bulbous bow and bare hull form sectional area curves generated earlier.

Forward fairness curve design section.

Forward fairness curve components split and embedded in the forward hull form.

Transverse curves. Bow curve at left. Transom curve to the right. Here we have elected to show the final bow profile, but note that there is a secondary design curve forming the outline of the aft portion of the bulb where it meets the stem.

Final output of the form parameter process: A multi-surface complex hull shown cutaway about the center-plane line of symmetry.
12.17 Randomly Generated OSV 3.1: Isometric lines view. Note that this hull has issues with fairness to the bulb, as shown in figure 12.21. This is fixed in OSV 4, as shown in figure 12.29.

12.18 Randomly Generated OSV 3.1: Side profile showing buttock lines.

12.19 Randomly Generated OSV 3.1: 3D view showing geometry from bulb to midship.

12.20 Randomly Generated OSV 3.1: Geometry from bulb to midship.

12.21 Randomly Generated OSV 3.1: Hull bulb connection with sub-optimal continuity conditions specified.

12.22 Randomly Generated OSV 3.1: Bow view showing curvature to the deck geometry from bulb to midship.

12.23 Randomly Generated OSV 3: Transition to Bulb controlled by bow fairing curve geometry from bulb to midship. Compare with figure 12.24, which has improved tuning using alternate continuity given by the procedures detailed in section 11.3.

12.24 Randomly Generated OSV 4: Transition to Bulb controlled by bow fairing curve. Smoothness between lower bare hull and bulbous bow, and from lower bare hull to upper primary hull, is controlled by tuning the bow fairness transverse curve. Compare this with figure 12.23, where boundary conditions encourage curvature to bunch up in parts of the surface connections. Here is this figure, we achieve better control using the procedures outlined in section 11.3.


12.26 Randomly Generated OSV 3: Transition to Bulb controlled by bow fairing curve (to much bulbous area in the bow fairness transverse curve in OSV 3). Compare with figure 12.25.

12.27 Randomly Generated OSV 3: Another shot of the transition to bulb controlled by the fairing curve. Transition from primary lower forward hull to upper forward hull near the bulbous bow has developed a notch. This can be tamed by adjusting the transverse bow fairness curve using procedural logic to ensure the fairness curve contains a small transverse bulb of sectional area comparable to, and slightly greater than, the bulbous bow section area. Note that the transition from the lower primary forward hull to the bulbous bow is quite smooth. Compare this with figure 12.28.

12.28 Randomly Generated OSV 4: Transition to Bulb controlled by the transverse-bow-fairness-curve leaving degrees of freedom open to handle continuity between upper and lower primary hull surfaces. The transition from lower surface to bulb remains good. Compare this with figure 12.27.

12.29 Randomly Generated OSV 4: Buttock lines at the triple transition flow smoothly to the cusp thanks to form parameter control of the transverse bow fairness curve, and control of continuity at the bow, and also the control of continuity to the bulbous bow. Compare this with figure 12.11.

12.30 Randomly Generated OSV: Lines Plan. Yet another randomly generated variation of OSV lines. See also surface and curvature plot in figure 12.31.

12.31 Randomly Generated OSV: Hull surface and Gaussian Curvature for the hull detailed with lines in figure 12.30 with curvature tolerance = [−.05, .05].

12.32 Randomly Generated OSV: Lines Plan. Randomly generated variation of OSV lines. Properties given in table 12.4 Surface and curvature shown in figure 12.33.

12.33 Randomly Generated OSV: Hull surface and Gaussian Curvature for the hull given in figure 12.32. Here curvature tolerance = [−.05, .05].

x
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Principal Inputs for the Bare Hull FPD System</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>SAC Form Parameters</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>DWL Form Parameters</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>CPK Form Parameters</td>
<td>47</td>
</tr>
<tr>
<td>7.1</td>
<td>Form parameters of initial SAC curve, Figure (7.1)</td>
<td>81</td>
</tr>
<tr>
<td>7.2</td>
<td>Form parameters of SAC curve with flat portion</td>
<td>81</td>
</tr>
<tr>
<td>7.3</td>
<td>Form parameters of SAC curve with flat portion. Centroid of area shifted.</td>
<td>82</td>
</tr>
<tr>
<td>7.4</td>
<td>Relative form parameters allow the solver a prescribed amount of freedom to position the stern bulb.</td>
<td>84</td>
</tr>
<tr>
<td>7.5</td>
<td>Form parameters for curve in Figure 7.4a</td>
<td>86</td>
</tr>
<tr>
<td>7.6</td>
<td>Form parameters for curve in Figure 7.4b</td>
<td>86</td>
</tr>
<tr>
<td>8.1</td>
<td>Some Experimental Input Design Space Specification Choices</td>
<td>136</td>
</tr>
<tr>
<td>8.2</td>
<td>Some Experimental Input Bulbous Bow Design Space Specification Choices</td>
<td>138</td>
</tr>
<tr>
<td>11.1</td>
<td>Aft Longitudinal Curve Form Parameters</td>
<td>206</td>
</tr>
<tr>
<td>11.2</td>
<td>Lower Forward Hull, Longitudinal Curve Form Parameters</td>
<td>209</td>
</tr>
<tr>
<td>11.3</td>
<td>Upper Forward Longitudinal Curve Form Parameters</td>
<td>210</td>
</tr>
<tr>
<td>12.1</td>
<td>Hull Design Randomly Generated Hull Form parameters</td>
<td>248</td>
</tr>
<tr>
<td>12.2</td>
<td>Hull Design Randomly Generated Hull Form parameters</td>
<td>249</td>
</tr>
<tr>
<td>12.3</td>
<td>Hull Design Randomly Generated Hull Form parameters</td>
<td>250</td>
</tr>
<tr>
<td>12.4</td>
<td>Hull Design Randomly Generated Hull Form parameters for lines shown in figure 12.32. Also, rendered surface and curvature shown in figure 12.33.</td>
<td>251</td>
</tr>
<tr>
<td>12.5</td>
<td>Hull Design Randomly Generated Hull Form parameters; center profile keel curve integrated with Bulbous bow</td>
<td>252</td>
</tr>
</tbody>
</table>
Listings

5.1 Basic Python Automatic Differentiation class .................................................. 68
5.2 Instantiation of AD numbers in 2D design space ........................................... 69
5.3 Instantiation of AD numbers part 2: Setup for use with Python’s Numpy Package .... 72

8.1 Implementation of the interval analysis class “ia” with some basic operations shown .... 114
8.2 Instantiation of and Extended Division with IA Variables ................................. 116
8.3 Goal Class ........................................................................................................ 120
8.4 States: A type of Python dictionary which maps logic variables to values. Following the design pattern of miniKanren, we equip our state with methods to bind variables to state, and to assign values to those variables. The practice in logic programming is to return a new state when binding and assigning new variables. ................................................................................................................. 121
8.5 state method, value_of looks up the value of a variable in an environment. ........... 122
8.6 “unify” : A Unification Algorithm Extended for Interval Arithmetic ...................... 125
8.7 eq algorithm to attempt unification and return unified (or nullified) state. Operator overloading of allows this method to be accessed with a call to Python’s == method. ........ 126
8.8 Usage of the == goal ............................................................................................ 127
8.9 “mullo” a Relational Rule For Multiplication ...................................................... 143
8.10 “binop” function for processing relational algebraic rules .................................. 144
8.11 Run Binary Relation: A function to extend binary relations to ternary .................. 146
8.12 Overloading Addition: relational on lists of states, member of the States class ....... 147
8.13 Addition syntax .................................................................................................... 147
8.14 addo, member of the Rules class ........................................................................ 148
8.15 Addition ................................................................................................................ 149
8.16 Overloading Division ........................................................................................... 149
8.17 Division syntax .................................................................................................... 149
8.18 Division syntax .................................................................................................... 150
8.19 Arc Consistency loop enabling non-local propagation of interval narrowing data .... 152

9.1 “Pretty-Printing” the computational graph for a Cb relation, using the PStates __str__ method .......................................................... 160
9.2 Pstates Class: Code for Compiling Python Rules into little logic language rules ....... 164
9.3 Pstates Class: Code for Compiling Python Rules into little logic language rules ....... 165
9.4 Pstates Class: Code for Compiling Python Rules into little logic language rules ....... 167
9.5 Pstates Class: Code for Compiling Python Rules into little logic language rules ....... 168
9.6 Pstates Class: Compiling Python Rules into little logic language rules. ................. 169
## Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{CP}$</td>
<td>Centerline Area</td>
</tr>
<tr>
<td>$A_{CPb}$</td>
<td>Bulb Centerline Area</td>
</tr>
<tr>
<td>$A_M$</td>
<td>Midship Area</td>
</tr>
<tr>
<td>$A_{Mb}$</td>
<td>Midbulb Area</td>
</tr>
<tr>
<td>$A_{WP}$</td>
<td>Waterline Area</td>
</tr>
<tr>
<td>$A_{WPb}$</td>
<td>Bulb Waterplane Area</td>
</tr>
<tr>
<td>$B$</td>
<td>Beam</td>
</tr>
<tr>
<td>$B_{b}$</td>
<td>Bulb Beam</td>
</tr>
<tr>
<td>$C_{ab}$</td>
<td>Bulb Centerplane Area Parameter</td>
</tr>
<tr>
<td>$C_{abt}$</td>
<td>Bulb Maximum Cross Sectional Area Parameter</td>
</tr>
<tr>
<td>$C_B$</td>
<td>Block Coefficient</td>
</tr>
<tr>
<td>$C_{bb}$</td>
<td>Bulb Breadth Parameter</td>
</tr>
<tr>
<td>$C_{Bb}$</td>
<td>Bulb Block Coefficient</td>
</tr>
<tr>
<td>$C_{Am}$</td>
<td>Bulb Midbulb Area Coefficient</td>
</tr>
<tr>
<td>$C_{CP}$</td>
<td>Centerplane Area Coefficient</td>
</tr>
<tr>
<td>$C_{CPb}$</td>
<td>Bulb Centerplane Area Coefficient</td>
</tr>
<tr>
<td>$C_{LB}$</td>
<td>Length to Beam Ratio</td>
</tr>
<tr>
<td>$C_{LCG}$</td>
<td>LCB to $L_{WL}$ ratio</td>
</tr>
<tr>
<td>$C_{lpr}$</td>
<td>Bulb Length Parameter</td>
</tr>
<tr>
<td>$C_M$</td>
<td>Midship Area Coefficient</td>
</tr>
<tr>
<td>$C_P$</td>
<td>Prismatic Coefficient</td>
</tr>
<tr>
<td>$C_{vpr}$</td>
<td>Bulb Volume Parameter</td>
</tr>
<tr>
<td>$C_{WP}$</td>
<td>Waterplane Area Coefficient</td>
</tr>
<tr>
<td>$C_{WPb}$</td>
<td>Bulb Waterplane Area Coefficient</td>
</tr>
</tbody>
</table>
$C_{zb}$ Bulb Depth Parameter

DLR  Displacement to Length Ratio

D    Draft

$D_{bulb}$ Bulb Depth

LCB  Center of Buoyancy

$LF_{CPK}$ Length of the Flat of Centerplane Curve

$LF_{WL}$ Length of the Flat of Waterline Curve

$LF_{SAC}$ Length of the Flat of Sectional Area Curve

$L_{WL}$ Waterline Length

$L_{bulb}$ Bulb Length

$L_{E}$ entrance Length

$L_{M}$ midship Length

$L_{R}$ run Length

$q(t)$ B-spline curve, parameterized by $t$

$V_i$ B-spline control vertices

$\nabla$ Displacement

$\nabla_{Bulb}$ Bulb Displacement

$x$: The interval, a pair $[a, b]$ of floating-point numbers representing an interval where $a \leq x \leq b$.

$C_{faM_{bulb}}$ Bulbous bow to Forward Fairness Area Coefficient

$C_{fa_{Lbulb}}$ Bulbous bow length to forward fairness station location Coefficient

$CASHD$ Computer Aided Ship Hull Design

$D_t$ Transom drop (non-vertical transom panel)

$HC3$ Hull consistency algorithm 3, arc consistency by domain decomposition and forward backward propagation.

$HC4$ Hull consistency algorithm 4, forward backward propagation directly on the syntactic tree.

$N_{i,k}(t)$ B-spline basis functions, parameterized by $t$

$THB$ truncated hierarchical B-splines

$X_{LCF}$ Longitudinal Center of Flotation

CPK  Center Plane Keel Profile Curve

DWL  Design Water Line Curve

SAC  Sectional Area Curve
Abstract

This thesis introduces a new methodology for robust form parameter design of complex hull form geometry via constraint programming, automatic differentiation, interval arithmetic, and truncated hierarchical B-splines. To date, there has been no clearly stated methodology for assuring consistency of general (equality and inequality) constraints across an entire geometric form parameter ship hull design space. In contrast, the method to be given here can be used to produce guaranteed narrowing of the design space, such that infeasible portions are eliminated. Furthermore, we can guarantee that any set of form parameters generated by our method will be self consistent. It is for this reason that we use the title feasible form parameter design.

In form parameter design, a design space is represented by a tuple of design parameters which are extended in each design space dimension. In this representation, a single feasible design is a consistent set of real valued parameters, one for every component of the design space tuple. Using the methodology to be given here, we pick out designs which consist of consistent parameters, narrowed to any desired precision up to that of the machine, even for equality constraints. Furthermore, the method is developed to enable the generation of complex hull forms using an extension of the basic rules idea to allow for automated generation of rules networks, plus the use of the truncated hierarchical B-splines, a wavelet-adaptive extension of standard B-splines and hierarchical B-splines. The adaptive resolution methods are employed in order to allow an automated program the freedom to generate complex B-spline representations of the geometry in a robust manner across multiple levels of detail. Thus two complementary objectives are pursued: ensuring feasible starting sets of form parameters, and enabling the generation of complex hull form geometry.

Keywords – automated form parameter design; constraint logic programming; artificial intelligence; reliable ship hull generation; truncated hierarchical B-splines

\footnote{“Robust” in the sense that the optimization does not fail - because we avoid infeasible problems to begin with. This is in contrast to “robust” in the sense of low sensitivity to changes in input, though these can be related.}
Chapter 1

Introduction

Motivation

Decisions made in the early stages of design have great effect on the lifetime costs of a vessel. Yet at this early stage the design engineer has the least amount of information to work with in developing the design of a new ship. This “knowledge gap,” between the data on hand and the data needed to make the best decisions, is the source of additional costs associated with the vessel, from fuel consumption to structural fatigue, and from cargo carrying capacity to construction costs and downtime for maintenance or even potentially how long the vessel must remain in port due to weather. The shape of the ship’s hull is one of the most critical and defining factors which influence all dimensions of its production and performance. Furthermore, hull shape is locked-in early during the design process, simply due to this great inter-dependency. Thus, early stage design, particularly of the hull form, is critical to performance of all systems associated with the vessel.

The good news is that at the earliest stages of a ship’s design, when design decisions are most critical to lifetime performance across all metrics, design changes are at their absolute cheapest. No steel has been cut. No detailed designs have been painstakingly drafted. A change in the hull shape does not entail the massive follow on changes that would ripple through the systems later in the design cycle, or, in worse cases, during production itself. Already we see that early stage design exploration may be paramount for economic viability of the ship design firms of tomorrow. A firm that is very good at early stage ship hull design will have a decided advantage over those that are not as skilled.

There is an additional reason for maximizing the accumulation of knowledge in the early stages. When ships are designed, typically the design is either one-of-a-kind, or one of a very few. This has important economic ramifications that can be seen by contrast with other industries where designs are mass-produced.
Figure 1.1: From Abt and Harries, [1], Cost vs. Time, top, and Knowledge vs. Time, bottom. This represents the knowledge gap. Expenses fixed by design decisions become large shortly after a project commences, but the knowledge accumulated is at a minimum here. Therefore, we strive to increase the amount of knowledge early on in the design phase, and so close the gap.
For example, in the automobile and commercial aircraft industries, the cost of development can be amortized across a potentially voluminous production cycle, and if a design flaw shows up in early models, the cost of fixing the issue is again spread across the production volume. Not so in the ship building industry, where a design run may consist of only one or at most a handful of ships. If an initial choice turns out to be sub-optimal, there is often no chance to get it right.

With these issues in mind, since the middle of the twentieth century researchers have sought the use of computers to aid in the ship hull design process. This has given rise to the discipline of computer aided ship hull design (CASHD). The hope is that a computer might generate a smooth and aesthetically pleasing hull which would meet design requirements while also providing a model on which design calculations and analysis might be performed. Combined, analysis and hull generation open the way to computational shape optimization. By analyzing many design trade-offs in the early stages and increasing knowledge of the design space, ship designers can make better choices and lower the risks posed by design mistakes.

Simply put, to reduce the risks posed by the knowledge gap, it is important to model and evaluate many hull forms quickly and efficiently during the early stages of the hull design process. Only by design space exploration can we map changes in hull shape to changes in hull performance. A major bottleneck in the process of evaluating many designs, and thus of optimizing effectively for a given set of design constraints, is the generation of many design alternatives across the relevant design space. By standard methods, hull design is an engineering-hour-intensive process of manual geometry manipulation. We seek to change this paradigm. Ideally, hull design generation would begin with a small and flexible set of inputs covering basic ship parameters such as overall dimensions and some minimum of other associated design parameters relevant to the particular design problem at hand. The algorithm would then take this basic problem definition and explore the space of solutions. The question now is this: “how do we automate this process of design exploration?” In this paper, form parameter design offers a key part of the answer.

“Form parameter design” (FPD) is one of the most promising methods being used to automate the geometric design of complex hull form geometry. In this technique, complex shapes are computed to user specifications directly as the solution to optimization problems rather than approximating user specifications

---

1 Including especially hydrodynamic (resistance) analysis, and possibly stability or motions analysis which might be performed using basic design and estimated mass properties. Furthermore, sensitivity computations for any of the above might be desired as well.

2 Further processing may be required to construct an approximate “meta-model” of the response space, in recognition of the fact that we cannot explore designs exhaustively (nor would we want too). After all, design spaces are exponentially complex in general.

3 Looking further ahead and into the design spiral, this model could serve as a basis upon which to build up a detailed design for ship construction purposes.
by manual adjustments to the design. This involves the use of high level descriptors for the properties the
geometry should take, and thus paves the way for systematic variation of the geometry across many parameters,
as needed for optimization. For example, by systematic variation of design parameters, relationships between
geometric cause and hydrodynamic effect can be established.

Form parameter design is not new. There are some well-known form parameter driven design systems
commercially available to ship designers, including the “Friendship” system, now a part of the CASES
framework at DNV/GL, and the Paramarine system now marketed by the QinetiQ corporation. The essential
methods of FPD have been demonstrated and the business case is solid; nevertheless uptake is slow across
the ship design industry. To some extent the reason for slow growth in usage is that, despite the enormous
potential benefit to be derived from early stage design automation, automation/optimization tools have a
reputation for being “hard to use.” This reputation seems to derive from user difficulties in getting form
parameter tools to generate viable geometry is because of the great complexity of the ship hull form itself.
This complexity can manifest as a difficulty for the designer in finding consistent sets of form parameters.

In this thesis the focus is on making truly automated form parameter design tools that work reliably for
complex hull forms and ensure robust performance in response to user inputs. Furthermore, we take note of
the fact that form parameter design is not “feature complete” for design space exploration. What is desired is
a tool that will “take the basic problem definition and explore the space itself.” Traditional form parameter
design supplies a method for generating one design only based on initial form parameter data. It is up to the
designer to supply the necessary starting form parameters and ensure they are workable. Similarly, a separate
program would need to supply a list of designs, each composed of a vector of starting parameters, in order to
use the basic form parameter tool to explore the design space.

Perhaps there is no surprise in stating that these issues of user-friendliness and supplying good initial
form parameters are closely linked. The reason that form parameter design programs are hard to use is that
the user must be an expert at picking form parameter combinations such that the program produces viable
geometry. In fact, it is the ease of picking inconsistent form parameters that causes a novice frustration in
using form parametric tools. This issue stems from the fact that form parameter design proceeds by numerical
optimization. If any of the constraints are inconsistent (mathematically infeasible) when issued in concert
with any other, then one or more of the internal optimization problems used to generate the design to the
parametric specification is bound to diverge. Furthermore, even if all the constraints are consistent, they may
still combine to give non-nonsensical answers, or geometry that which meets the constraints in a manifestly
A quick note about the methods that will be employed is in order: Often form parameter design papers focus on small variations of parameters. Or they abandon equality constraints as these are deemed too constrictive and may lead to non-physical solutions, or singular systems of equations, instead of ship hulls [3]. In this case relaxation to inequality constraints makes the problem of finding consistent sets of form parameters easier, but this is not the approach we follow here. Instead, using techniques from other areas of computing, namely interval analysis together with constraint logic programming, we demonstrate a program which solves the issue in a more direct and hopefully robust manner. With such methods in hand, we can then do form parameter design with equality constraints, and perform ab initio design with confidence.

Towards Robust Form Parameter Design

In [4], Volker Bertram notes that “Design optimization problems require in most cases tailor-made models, but the effort of modifying programs is too tedious and complex for designers. This is one of the reasons why optimization in ship design has been largely restricted to academic applications.” Here methods of ‘machine intelligence’ may help to create an algorithm for each individual design problem. He goes on to suggest that three ingredients have to be supplied by the designer:

- specified quantities,
- unknowns including upper and lower bounds and desired accuracy, and
- the applicable relations.

Standard software approaches, even sometimes with form parameter design, do not include this flexibility of problem definition. In the absence of more powerful and abstract approaches, designers may find themselves adapting their solutions to fit the tools instead of the other way around.

In hull design, this can manifest itself as the parent hull design philosophy. The rough draft of the “solution” has already been found, in this philosophy. The designer takes some preexisting hull form and modifies until it meets the basic requirements for the new design. This often involves tedious manual manipulations of the control mesh, or perhaps another method using small deviations or extrapolations from the starting point. This is an acceptable solution because it is very time-consuming to develop a fresh new hull form by trial and error by ab initio manual design. Such difficulty leads engineers to conclude that it is better to force an old solution into a new shape than to attempt anything genuinely new from scratch.
This thesis will adopt Volker Bertram’s idea — that the expert system, the optimization shell, or the knowledge based system, as it is alternatively named, “may help to create a suitable algorithm for each individual design problem.” In this thesis this means that much of the data supplied by the user can actually change the code which turns the rules — relationships between geometric constituents of the hull — into computable facts which enforce those truths in the program, and are then used to process the design space itself. Before going too far into the details on this, we should note that this type of program development has been attempted before, but success with design problems has been marginal. Several reasons for this seem to have been to blame [5]:

- “Design is often a creative process with few documented rules.”

- “Human ‘experts’ have difficulties describing how they proceed in design.” “The problem can usually not be decomposed into independent sub-problems.”

- “Different ‘experts’ may also proceed very differently coming to very different designs (with no consensus which design is ‘best’)”

- “Design often involves extensive numerical analysis and computer graphics.”

As shown in the literature review, most of these attempts came before form parameter design blossomed into its current form in the late 1990s. If we assume form parameter design can offer an answer to the above set of objections, then we might amend the list above to instead say the following:

- Well defined automated procedures for working with spline based geometry representations really did not exist at the time when research and development of expert systems and logic programming was at its height. Form parameter design had yet to be modernized to work with B-splines and hull form design on the computer was typically an interactive process. There was no parsimonious way to encode the knowledge of hull design experts because of the primitive state of computational graphical design systems.

- Geometry is continuous, and there has always been contention as to the best way to handle continuous quantities in logical programming paradigms. Of course, with production systems, i.e. collections of if then rules, real valued information can be handled with inequalities, but the application of if then rules to form parameter design seems somewhat cumbersome.
In fact, one might reasonably argue that not many naval architects, nor other “physical engineering” practitioners are very familiar with the principles of logic programming, and so called “production systems” with their simple if-then rules. Application of such rigid technology to design problems seems destined for the same issue mentioned by Bertram above — that the program could not be easily adapted to the optimization problem.

So, part of what will be done in this thesis is to put form parameter design together with logic based programming. It may not yet be obvious, however, that the two should have been used together from the start in ship design. After all, expert systems require rules from experts, but we propose an automated system for design space search.

We must also deal with the fact that both hull geometry and design spaces are continuous. We need a way to express continuous relationships between the form parameters of our design space. Part of the solution comes from basic principle of naval architecture: the design ratios provide the essential method of relating various components of a hull design together. These will be the basic design space constraints in our constraint logic programming solution. Naval architects naturally think of design ratios as expressing relationships between continuous parameters of the design. That leaves the issue of dealing with real numbers in the logical programming paradigm.

Luckily it turns out that interval analysis naturally provides just the kind of capabilities that we need to augment a traditional constraint logic system to be able to handle real numbers naturally as intervals. Interval techniques are a set of numerical procedures in which intervals replace real numbers, interval arithmetic replaces real arithmetic, and interval analysis replaces real analysis [6]. A stated by Benhamou in [7], the intervals have been incorporated in a number of augmented Prolog systems to allow a natural processing of relational intervals constraints. They use fix-point algorithms, the simplest being something akin to arc-consistency. We will do the same here, but in Python, in order to closely couple this design space reasoning system to the rest of the form parameter design program.

Now, with form parameter design together with a minimal inference engine based directly on the basic algorithms of declarative logic programming, and rules consisting of basic design ratios and similar expressions, together with interval arithmetic, a method of ensuring robust form parameter design can be demonstrated. Another way of looking at this is that form parameter design is a methodology that can be used to extend expert systems into the realm of complex hull modeling, while on the other hand, constraint logic programming can offer robustness and natural constraint management to form parameter design.
One could think of the logical constraint programming component of the system as a global filter on the design space, quickly removing infeasible portions. Then it is natural to also look for local methods of ensuring robust performance of the system. Here we employ the truncated hierarchical B-splines to allow for particularly structured local refinement when solving for curves. The particular structure will make an especially nice way to incorporate the local features into the final hull surfaces.

**Contributions**

The chief product of this thesis is a clear methodology by which to ensure robust form parameter design. In particular, complex hull forms can be accommodated, and the method can flexibly allow for designers working on new problems to come up with unique solutions. The strength of the method is the flexibility of allowing for user specified constraints, together with automated constraint management.

In pursuit of this goal, we make use of the unification algorithm and the arc consistency algorithm from computer science, together with interval arithmetic. These methods are brought together using an internal domain specific language construct that allows for the designer to enter rules into the system with very simple expressions. These methods are used to develop a constraint consistency system which acts as a filter directly on the design space of a hull design problem, eliminating infeasible subspaces from consideration automatically.

Further, once designs are chosen from the feasible domain, the form parameter design system is itself augmented by being developed via automatic differentiation. The result of this construction is that the form parameter design system can model a particular design problem in a “functional” and “composable” fashion. A potential extension to the work presented here might further increase the flexibility of the system by using this composability to its fullest, and letting the logic programming system also dictate exactly how the variational problems of form parameter design are constructed.

Continuing on, the solution phase is augmented by allowing the B-splines to be represented in the THB-basis in order to robustly solve for local details and to pass them onto the final hull surface without undue growth in the number of control points.

There is novelty in using all of these methods together and novelty in the aims of this project. This implementation must be robust to local design peculiarities and at the same time efficient in trimming

---

4Functional in the “programming paradigm” sense. —Of course the optimization problems are themselves “functionals” but this time in the “to be minimized” sense!
infeasible designs \(^5\) from the global search space, and making itself easy for a new user, experienced naval architect, or another computer program, to use.

So original work is found in applying interval, automatic differentiation, and relational methodologies to the naval architecture problem of ship hull design. Additionally, the B-splines are extended to the topologically more flexible THB-spline and these are used to construct an adaptive curve solver and compatible surface definition. The primary new features are to add the capability to reason from a hull design specification to a consistent set of form parameters, to automatically infer when a constraint set is infeasible, to speed the process of form parameter design by narrowing the search space to bounds of the feasible domain, and to provide an adaptive solver capable of complex geometry.

A portion of this thesis appears in the following publication:


The above publication deals with automatic differentiation and form parameter design. The majority of this thesis is unpublished work, especially the sections incorporating interval analysis, declarative constraint logic programming, and the truncated hierarchical B-splines.

**A note on Geometric Implementation:**

In form parameter design, B-splines and NURBS are again employed as the basic underlying representation for geometry. The Lagrange multiplier method mentioned above actually serves to transform the underlying control vertices of the B-spline curve. The optimization algorithm operates directly on these points. As such, the form parameter driven B-spline curve design algorithm developed here can easily be applied to NURBS curves by switching from 2D/3D coordinates to their homogeneous equivalents. However, changes in weights and shifting of vertices have similar effects on the shape of NURBS curves. This ambiguity lessens the efficiency of using both vertex coordinates and weights as free variables in the curve optimization. Therefore, modification of weights is not considered here. For the same reason, only uniform knot vectors for open curves (specifically, uniform away from the endpoints) are used and the knot vectors remain unchanged during the optimization. Please note that the knot vectors are of “endpoint interpolating” type. The curve design procedure is implemented in the well-known interpreter language Python [8].

---

\(^5\) infeasible design (sub)spaces
A bit of a technical nit comes up here. In the truncated hierarchical B-spline literature, any knot vector which is not uniform, even if only at the start and ends, is “non-uniform.” This is of particular importance for the wavelet projection operators that scale from one level in the hierarchy to the next.

Also, a paper on optimization featuring code written in Python would be remiss not to mention the scipy.optimize package, which contains numerous sub-packages useful for gradient driven optimization, or similar. Since part of the purpose of this paper was to show how AD works, and how it can be applied to B-spline curve design in a very explicit manner, the scipy.optimize package was not used. Instead, the NumPy linear algebra package was used to solve the resultant linear systems of equations after the AD routines had done the work of generating the system, as shown in sections 5 and 6. A similar philosophy was followed for the interval class, constraint logic programming portion, and truncated hierarchical splines.

It may also be noted that modified Newton algorithms, such as sequential quadratic programming (SQP), or optimized second order algorithms such as Broyden-Fletcher-Goldfard-Shanno (BFGS) or quasi Newton methods, could be used to lessen the burden of computing the full Hessian matrix, or to take advantage of existing optimization libraries. It must be noted that part of the purpose of this paper is to show how AD can speed development of new software for the automated generation of geometry of engineering interest. Optimizing solver performance is not a goal. Optimizing programmer performance, and finding better curve generation methods, not necessarily faster, is the goal. Only once those objectives are met, or inhibited by solver performance constraints, will developer time be spent on squeezing more performance out of solver routines.

**Final Note**

In this thesis, almost all curves and surfaces are designed by form parameter design using inputs generated randomly and consistently by computer. The exceptions are the curves given in section 7, which are meant specifically to demonstrate automatic differentiation of B-splines, and the cartoon form parameter curves of figure 3.2.
Chapter 2
Related Work

In this section, we give a history of hull form design including form parameter design. Additional background is supplied for the supplementary methods to be used in this thesis. In order, we will discuss the following:

- Hull design
- Geometry representation including B-spline and related technology up to and including the truncated hierarchical B-splines
- Form parameter design
- Automatic differentiation
- Interval arithmetic
- Constraint logic programming

Hull Design

Research in the field of form parameter driven curve and shape design goes back many years. In fact the first ships lines plans date from around 1700, as noted in [9], [10]. Even though lines plans have existed as representations of vessel shape for some time, designers have preferred to use practical methods, such as battens and weights, to control the shape of hull surfaces, at least until computer aided design (CAD) tools were developed and desktop computers became affordable and effective [10].

1 Lines plans served as documentation of empirically developed ship hull shapes. Chapman, [11], [12], pioneered the use of families of parabolas for waterlines and other hull shape curves and even mentions the design spiral, mentioning "retouching the body, and of again going through every part of the calculation."
Mathematical representations for hull surfaces were originally constructed to facilitate hydrodynamic analysis, especially the work of Taylor, as in [13], where he used polynomials to assist in hull form development in that a parent design could be systematically varied to evaluate the change in resistance with respect to change in various form parameters. However, as noted by many, see for example, [14], and [10], such an analytical approach to hull specification is often subject to fluctuations and polynomial representations can be non-intuitive for direct manipulation.

Alternative representations have been based on conformal mapping [15] [16], or offsets [14]. Finally, Kuiper [17], using a line of research initiated by Taylor, developed a system for composing longitudinal curves of from, that is, the waterline, profile, and sectional area curve, from an initial set of hull characteristic parameters. Kuiper also distinguished between “design parameters,” such as length, beam, longitudinal center of gravity (LCG), etc., and “form parameters,” which are used to directly compute basic curves. Similarly, Buczkowski, [18] developed a parametric approach, notable for the appearance of a fairness function, and the minimization of a fairness functional, a mechanic that has proved foundational for form parameter design approaches. However, this approach was still limited by the mathematical representation itself. Nowacki and Reed, in [19], made use of conformal mapping for modeling hull shape below the design waterline, while polynomials were used for shape control above. This extended the possibilities for shape representation.

**Computer Aided Ship Hull Design**

Throughout the second half of the twentieth century, computer systems were rapidly increasing in performance, and new forms of geometric representation were being developed, especially the bezier curve [20], and the extension to the B-spline with [21], [22], and [23] and finally the rational extension of these with NURBS [24]. These improved representations could not only represent essentially arbitrary geometry\(^2\) but the manual manipulation of that geometry was much more intuitive than with the old polynomial or conformal representations. This greater ease of use meant that CAD designers could now work more efficiently manipulating the curves directly than in working with more abstract\(^3\) design tools. This new paradigm of interactive curve creation and hull form manipulation was detailed in [25] and [19].

\(^2\)With the caveat that in standard formulations the control mesh itself is topologically “rectangular” — see any computer aided graphic design text for details.

\(^3\)e.g. parametric
Geometry Representation

Through the extensive growth of the paradigm of interactive geometric design, B-splines became the standard of computer graphics\(^4\) and especially computer aided design systems. This subsection will briefly outline recent developments in geometry representation technology since it has much significance for the ship hull design and representation problem.

B-splines and NURBS curves have several nice properties that make them attractive:

- **Efficient Evaluation** (of both the geometry itself and its associated mathematical properties)
- **Compact Support**
- **Affine invariance**
- **Partition of Unity/Analysis Suitability**
- **Smoothness.**
- **Ease of evaluation of derivatives and integrals of the geometry.** (arc length, tangency, curvature, area, volume, centroids)

In engineering design, B-splines and their NURBS extension remain the standard to this day. However, they are not without competition.

**Competition from Subdivision Surfaces**

Subdivision surfaces, which are the top representation for geometry in more artistic domains such as computer animation, and which do not have the topological constraints of B-splines, have been constructed to meet fairing requirements and engineering constraints as well. Many such schemes and capabilities have been published. See [26] for one of the first schemes to propose fairing with Catmull-Clark surfaces, see Stam’s paper [27] for their efficient evaluation, and [28–30] for recent applications of subdivision surfaces to ship hull design with constraints. Despite these advances, a mitigating factor is that subdivision surfaces lose their nice computational, geometric, and continuity properties, and thus the accurate evaluation of some geometric constraints breaks down, at exactly those locations in the mesh which extend beyond the capabilities of the spline based methods. That is, at extraordinary points, subdivision surface analysis becomes at more

\(^4\)Though now superseded by subdivision surfaces for graphical applications such as animation.
conditional and problematic. For instance, often the connective continuity of subdivision schemes is C1 here. Closed form computations, such as for volume, break down as well [31]. So the catch is that the topological flexibility of subdivision surfaces is limited to evaluation, precluding full surface analysis and interrogation near extraordinary points. This limits the power of subdivision surfaces in the inverse design context, to something more akin to that of more structured B-spline representations.

Difficulties with full analysis, including integrals and derivatives involving extraordinary points, seem to be the primary technical limitation for subdivision methods in engineering modeling and design, however this is not the main reason for choosing B-splines over subdivision representations. B-splines remain the standard engineering representation due to their simple and efficient implementation, the great many programs in CAD design and engineering analysis which use them, and the simplicity of data transfer (as long as they are not trimmed). As such B-splines are employed in this thesis as the mathematical basis for curve design and representation of hull surfaces. Development of B-splines has been extensively covered in technical literature, see [32, 33] for the basic definitions and properties. An in depth reference covering definition, properties, and especially the implementation of algorithms for the manipulation of B-spline curves is provided by Piegl and Tiller in their book, [24].

Despite having pride of place as the standard in geometry representation for engineering purposes, B-splines have intrinsic shortcomings as well, especially when used for surface definition. These are as follows:

- A single NURBS patch is either a topological disk, tube, or a torus. In the standard computational representation the disk is a four sided surface.
- Multiple patches are needed to model complex geometry requiring patch management, gluing conditions, and the like. This adds complexity to simple processes like deforming the shape by moving control vertices, and also complicates constraint satisfaction.
- Non-local refinement. Adding a single control point requires an additional row and column of control points be added to the surface in order to maintain the regular mesh.

This issue of local refinement is important for the robustness of our form parameter design tool. An adaptive solver can theoretically find superior solutions to a non-adaptive one, and also find solutions in situations where the standard methods and representations would fail. Local refinement is a key requirement
for adaptability. Furthermore, local refinement eases the difficulty in maintaining patch-patch continuity conditions, or other such boundary requirements.

**B-spline Extensions for Local Refinement**

Various spline based techniques have been developed to address the local refinement issue without changing the basic spline paradigm “too much.” These spline schemes include hierarchical B-splines [34–37], T-splines [38, 39], and LR splines [40, 41]. Most of these extensions are forced to abandon some useful properties of B-splines/NURBS in order to achieve local refinement. For instance, the early versions of the hierarchical B-splines sacrificed the partition of unity and affine invariance properties, though as we will see, these shortcomings were later rectified.

Two essential branches can be seen forming in the spline literature. On one hand, subsequent refinements of the hierarchical idea have been ongoing, such as polynomial splines over hierarchical T-meshes [42], and truncated hierarchical B-splines (THB) [43, 44], spline forests, and truncated Catmul Clark subdivision [45]. On the other hand, T-splines have been further developed as well [38, 39]. Important variants include analysis suitable T-splines [46, 47], modified T-splines [48] and truncated T-splines [49]. A hybrid scheme can be seen with hierarchical T-splines [50].

THB-splines were selected for this project due to the relative elegance of their formulation over that of the T-splines. This amounts to a heuristic choice made with an eye towards managing the complexity of implementation. Truncated hierarchical splines (THB-splines) were introduced by Giannelli, Juttler, et. al in [43]. The THB-splines are derived from the original hierarchical B-spline work of Forsey and Bartles [34], who first described a multi-level B-spline system for hierarchical surface definition in which the fine levels overlapped the coarse levels. Rainer Kraft, [35], furthered the development of hierarchical B-splines by developing a global selection mechanism to select linearly independent basis functions from different hierarchical levels. The truncation mechanism adds the satisfaction of partition of unity to the list of properties that can be satisfied in the most well-developed hierarchical B-spline construction [43]. This last property is fairly important since affine invariance is lost without it, and without partition of unity, the curve is no longer confined to the convex hull [51].

This is enough spline and subdivision representational introduction to facilitate more development of the history of ship hull form generation attempts. We now proceed with the main developments in ship hull

---

5The end user will be the judge!
Geometry Engineering for Ship Hulls

Manual geometry manipulation became the standard in CAGD through the 1980s, but specialized methods to facilitate the generation and variation of ship hulls have been in use concurrently. The chief methods are as follows:

- **Lines Plan Definition**: Following standard naval architecture practice, this methodology does not target a mathematical surface representation [52].

- **Morphing/deforming existing hulls**: Subsets of the hull geometry, such as a selection of offset points, or curves, are targeted for systematic manipulation. Special techniques [53] sometimes come into play.

- **Interpolation from Standard Series**: This is the standard approach pioneered by Taylor, who used parameterize curves systematically deformed to generate a series of shapes. These can then be interpolated to obtain geometry (e.g. hulls) with properties that fall between two data points on the series.

- **Free form/manual curve and surface manipulation**: Here the designer manually and directly manipulates the curves and surfaces making up a hull shape. Ship hulls are defined by manually defined surfaces, which also serve to define the ship’s lines.

- **Form parameter design**: The hull is defined first by a set of form parameters. These define properties such as length at the waterline, depth, LCG, and so on, including physical characteristics as well, e.g. displacement.

In practice, many times various methods are used in concert. In the review below, we will discuss the methods with some sources which were defined various turning points in this long history.

**Free form manual development of hull geometry**

To develop a hull via manual geometry manipulation, an iterative design process is normally undertaken. If the hull is simple enough, a complete surface may be manipulated by direct manipulation of the defining control polygon. Though this technique seemingly allows great control, in practice very high skill is needed to obtain a hull form which satisfies desired characteristics of smoothness and also meets design parameters [10].
Greater control can be obtained in the manual domain via the adoption of structured approaches. A productive line of attack is to define control and definition curves from which the hull may be built up. This net of control curves can contain both basic location information which the surface will interpolate, and other curves may be designed to specify transitions between one surface region to another. In any case, manual manipulation of a hull surface, or its definition curves, is a time intensive process requiring expert care from the designer.

It might be noted that form parameter design tools actually automate and expand some of these approaches, especially the curve to surface representations.

**Hull Morphing, Parametric Shifting, and Systematic Variation or Distortion**

**Parameter Shifting** In the past, known good designs were highly expensive to modify. A process of parameter shifting was developed to automate simple property transformations of pre-defined hulls. Such processes minimized the work required by inheriting smoothness and other desirable properties from the parent hull. The work of Lackenby, [54] was for many years the standard for the parameter shifting style of ship hull design variation.

The methods are popular because they are simple, widely available, and well understood. Issues with such methods include the inability to constrain many features under transformation.

The method of interpolation from standard series is a different compromise based on the same general ideas of parametric shifts, but with the features more tightly controlled under transform, since the transforms are all interpolations between given designs. Despite the rigidity of these methods, they were effective compromises especially given limited computational power to work with.

**Hull Morphing** Free form deformation [53] (FFD) is an example technique for morphing a geometrical entity into new specific shapes. It is possible to preserve local information while making such deformations.

This is an alternate type of structured shape modification in which, for instance, a hull surface would be embedded in a geometric solid. Then, by manipulation of the solid, transformations of the surface may be obtained in which some of its properties may be held constant.

In this way, FFD techniques combine “free hand” techniques with mathematical transformational accuracy and control and merit continued interest as they allow the designer both control and precision in sculpting a hull shape.
**Parent Hull Methods** have remained popular in ship hull design. Following Keen and Tibbitts, [55], several reasons make parent hull methods attractive:

- They can avoid the scrutiny that a new clean sheet design receives from the many offices involved in the design process, usually in the form of additional approval milestones.
- It is argued that design and construction costs will be less.
- It is rationalized that construction can start earlier in the acquisition process.
- Parent hull method are an easier sell to conservative organizations such as the US Navy, thus central planners can sell the program more easily, promising reduced technical risks by only modifying a successful design.

Parent hull methods risk stagnation or conformity of design, if taken to the extreme. However, Kean and Tibbitts go on to note the key fallacy which government program offices and industry leaders put forth in settling for designs derived from parent hulls: “The design is mature”. This is often an unsubstantiated claim [55] that gives rise to further unsubstantiated beliefs. Common false conclusions drawn from the erroneous assumption that a design is mature are as follows:

- Schedule will be more compact than historical experience (production and development can be concurrent).
- Weight will not grow as usual.
- Affordability initiatives will reduce production cost.

Contrary to these beliefs, physics and geometry dictate that the hull form re-use or re-purposing of old designs is wasteful for several reasons [55]:

- Hull forms in all likelihood become less efficient when resized.
- Increased ship density to accommodate new mission profiles results in reduced stability and this in turn puts pressure on damage stability requirements.
- Structural analysis must be undertaken with equal rigor to a new design, yet we have self-imposed constraints that were unnecessary due to the fact that this is a modified old design. This implies a weight penalty.
• By ‘pushing the envelope’ with an existing design, the possibility of exposing latent failure modes is increased. This offsets the perceived ‘extra risk’ of a totally new design.

• Further, by pushing an old design outside of its intended parameters, we may actually increase the systemic risk compared to relatively unconstrained engineering using up to date methodologies.

The general principles are these:

1. The old design acts as an additional set of constraints on the new design.

2. Design properties inherited from the old design will propagate through the design process for the new design.

The authors go on to advocate several new rapid hull prototyping and early stage ship hull design frameworks, some of which share lineage with this current work as it regards form parameter ship hull design [55].

Next, we introduce form parameter design for ship hull design. This is the state-of-the-art system for hull form generation today. The goal of form parameter design is to exploit automation to perform design space exploration and optimization with the goal of producing the best designs possible.

**Form Parameter Design**

Conceptually the form parameter design method transforms a user specification into an optimization problem — so called “inverse design.” The software reads the user specification and returns the desired shape in the form of a mesh of control vertices. In general, the inverse design problem is nonlinear. It is solved here using Lagrange’s method of multipliers with a set of constraints, called the form parameters, and a set of fairness functions, which quantify the “goodness” of the shape in an objective manner. For background on the Lagrange multiplier technique in optimization, a multitude of sources are available, such as [56], [57], and [58].

A general overview of computational curve design as it has progressed over the recent decades is given in [9, 59, 60]. As mentioned above, from the time of Taylor’s research onward, parametric methods have been present in the naval architecture toolkit. However, as computational power has increased, and the creative and engineering limits of both deriving hulls from a parent, or the time constraints of developing hulls in free-form have come to seem ever more constraining, and especially when combined with massive online computing
resources and analytical tools for optimization, form parameter methods have become state-of-the-art in hull form design once again. For the methodological research and development of parametric hull form methods, see [9, 10, 52, 61–71]. These authors present methods for the design of complex, “fairness optimized,” ship hulls using form parametric techniques. In [9], procedures for formulating individual curve design as an optimization problem are laid out and the basic concepts of form parameter design are outlined which use curve optimization as the core tool. Some computational particulars are sketched especially for Bezier curves. Form parameter design is shown to proceed by development of “curves of form” defining aspects of hull shape across the entire hull, and then section curves instituting the global shape parameters as constraints at the hull sectional level. Harries [52], extends this for B-spline curves and CFD optimization. In [63], form parameter design is extended to complex hull topologies. Interest has continued into applications for particular hull form generation tasks, for example in [1, 3, 72–79]

In addition to the general criteria of hull design and analysis, form parameter design systems have addressed issues of feasibility, ease of use, and robustness as well. For this particular aspect of the problem, a wealth of methodologies have been tried. In the initial work of Harries, [52], a direct computational system is developed which uses the B-spline control polygon and known constraint properties to compute allowable ranges for the vertices. His feasibility reasoning utilizes the convex hull of a B-spline curve to estimate suitable starting points for a curve design optimization algorithm. For a simple example, consider a B-spline with an end tangent constraint where the tangent angle is defined to some known reference line, typically the $x$-axis. By construction, we know that a B-spline with an end tangent angle requirement must have its last two control vertices aligned such that a line passing through the two of them would be at the same angle with the reference as that of the tangent constraint itself. Constructions for end curvature requirements are similarly determined. Area and moment constraints are computed with simple polynomial expressions. Harries’ work makes assumptions about the number of control points but could in principle be extended. Algebraically derived extensions may not be general, but instead need to be derived depending on the type of spline or specific geometric constraint required. See [52] for details.

Notably, Harries’s feasibility computations involve the control points of only single B-spline curves. There is no computed constraint interplay between independent curves, although this is implicit in the hull form. In other words, curves may be solved independently, but all components of a hull’s shape are very much inter-related. Some simple facets of hull form design are long recognized as inter-related. These relations are codified in the form of various naval architecture hull form coefficients. For a local non-linear curve
solver, ignoring the implicit constraints between curves is advantageous only after the program has already established consistency between the curves and surfaces in some other way, or if the particular data in the particular curve is actually a representation of an inter-relationship on some other representative level of the model. This is needed because the success of a particular non-linear problem to design one curve is wasted effort when the curve is inconsistent with other facets of the design. Somewhere, these inconsistencies will likely manifest themselves in the failure of a some other curve solution attempt. Form parameter feasibility is not just a local problem, but one that extends across individual curve solutions.

Within traditional form parameter design, this kind of feasibility could be ensured by iterating over sets of form parameters to check for intra-curve (global constraint) feasibility. This could be computationally expensive.

At around the same time that Harries was developing his bounding box approach to local constraint feasibility, a complementary approach was being developed by a group of researchers who explored neural net and fuzzy logic approaches to the problem of ship hull form design. In [80], Lee et. al. used fuzzy modeling to generate SAC form curves for initial hull design. In [81], Kim and others extended fuzzy modeling for initial hull form modeling, including parallel mid body, bow and stern profile curves. Interestingly, they also note that form parameter design relies on “trial and error” to find consistent form parameters for a given design, and cite this issue as the reason why form parameter design has a steep learning curve for the inexperienced user. These models develop small adaptive neuro-fuzzy inference systems (ANFIS) to represent the hull form curves. In [82], the ANFIS system data is learned from parent hull data. Parent hull data is originally derived from B-spline approximation and genetic algorithms. A driving idea in these papers was to use the ANFIS net to store approximate data about known hull forms in order to infer parameters for generation of new hull forms. This does not directly address the general constraint feasibility issue of form parameter design but circumvents the issue by machine learning. The disadvantage is that the method requires new hulls be based on information stored in the system about other hull forms.

These fuzzy logic and neural network paradigms offer much in the way of resources to draw on for inference and search algorithms. Other researchers have used neural networks, fuzzy logic, and evolutionary or genetic algorithms for hull design [83]. See [84] for a summary as of 2009. Evolutionary strategies and genetic algorithms are often used to inform form parameter choice, while neural nets are used as proxies for the efficient approximate objective function space searching. These are search strategies that rely on resulting hull form evaluations to make educated guesses about which values of individual form parameters will be
both advantageous for the design objectives at hand. No direct attempt is made in these approaches, to rule out infeasible combinations of form parameters.\(^6\)

Finally, the state of the art in form parameter design, represented in the continued line of work from Harries and the Friendship framework (now CASES with DNV/GL) is to use meta-modeling approaches or design of experiments [85], and smart sampling (Sobol or similar) for efficient search of the form parameter design space, along with inequality constraints to aid in finding feasible sets of form parameters (by relaxing equality constraints), and coupled with multi-objective optimization and search algorithms using physical simulation tools of choice to fit the problem at hand [1], [73], [3], [4].

It should also be noted that some constraint management takes place through a design hierarchy [68], [69], [70], [4], [71] where information propagates from design specification to hull form parameters, to section and station wise form parameters, etc. This helps greatly in managing constraint complexity.

In all of these methodologies except the control vertex reasoning methods of [52], and in the hierarchical constraint generation architecture of the general form parameter system, feasible combinations of form parameters are searched for. They are neither inferred, calculated, nor deduced. The work of this dissertation will chiefly strive to extend the analytic and logical methods by which feasible form parameter sets are generated.\(^7\) We will use logical methods to contract the infeasible domain.

**Automatic Differentiation Related Work**

The necessary condition for a minimum of the augmented Lagrangian function commonly results in a system of non-linear equations. Solution of the system of non-linear equations requires extensive derivative calculations. In this paper the computation of derivatives of the Lagrangian are handled by automatic differentiation (AD). Not only does this reduce the programming burden during implementation of the method, it also provides greater flexibility in the selection of applicable constraints. As noted by Harries [52], the inherent disadvantage of form parameter design is that only those shapes can be generated which are defined by existing form parameters. As a consequence, the set of form parameters available to the system must be extensive enough to allow for a wide variety of design shapes. In the best scenario, a system should be flexible enough to incorporate new parameters as they are needed. Automatic differentiation facilitates this goal by simplifying implementation issues. Using automatic differentiation, the programmer need only

---

\(^6\) As such they may be complementary to the approach to be shown here.

\(^7\) Note for reference that a “feasible form parameter set” always represents a realizable hull form.
state the mathematical formula for computing a form parameter. The required derivatives are then computed without further programming efforts.

The technical development of automatic differentiation (AD) is described by Griewank in [86]. Christianson presents a review of the performance of “forward mode,” vs “reverse mod” AD in [87]. Applications can be found in [88–93]. A detailed review of the AD scheme employed in this paper is found in [94]. In [94] schemes are presented for first and higher order differentiation. The present paper restricts differentiation to first and second order, since that is all that is required for typical Newton’s method schemes of nonlinear optimization. The addition of AD will allow increased flexibility in the specification of objectives and constraints in our problem. In addition to easing the programming burden, this will allow for the specification and solution of more complex curves. Curves commonly specified in multiple sections can be done as a single unit, reducing the constraint management burden since some integral constraints would otherwise need to be specified across multiple integral regions, or boundary conditions would need to be satisfied at sectional curve ends.

The basics of automated differentiation (AD) are summarized in Section 5. AD packages for Python are readily available (see pyadolc [95], algopy [96], pycppad [97], CasADI [98], and ad [99]). Additionally, Python optimization and machine learning packages such as Theano have facilities for automatic differentiation as well [100]. We choose to implement our own version with an interface that maximizes interoperability with curve design utilities, and differentiates only desired computations. The automatic differentiation procedures are implemented in the interpreted programming language Python as well [8].

**Interval Arithmetic Related Work**

Interval techniques are a set of numerical procedures in which intervals replace real numbers, interval arithmetic replaces real arithmetic, and interval analysis replaces real analysis [6]. Theoretical numerical analysis considers exact quantities and exact functions acting to transform this exact data, while practical numerical analysis works with finite precision approximations of the ideal quantities. Interval arithmetic is a method of taking this imprecision into account.

Interval arithmetic [101] is widely known for giving bounds around floating-point calculations [102]. What might not be as immediately obvious about interval arithmetic is that, through extensions to interval analysis, it given a collection of very powerful tools for finding solutions to nonlinear systems of equations,
and for global optimization [6], [103], [104].

Moore, et al. in [102], introduce interval analysis and provide examples of its usage and utility in a variety of computational circumstances. Hansen and Walster [6], explain some general techniques for provable global optimization of nonlinear problems. Here, the focus is on iterative techniques, and good overviews of contractor algorithms are found. Jaulin et al. [104] give an overview of interval analysis with a focus on contractor and branch and bound “paving” techniques. Paving is their preferred term for the tiling of the search space in the quest for solutions to global optimization problems. Other books which follow the aim of introducing and explaining the wide-ranging techniques, variations, and applications of interval analysis include [101], [105], [103], [106]. Kearfott notes extensions to interval analysis, including its use in Prolog style logical programming over continuous domains [103], much as it is used in this thesis. See also Kearfott [107] for related work in decomposing functions, something that we will also address here. Two excellent papers summarizing interval capabilities with regard to optimization are one by Ratschek and Voller, ‘What can interval analysis do for global optimization?’ [108], and an introduction by Kearfott [109]. For a focus on implementation details and novel extensions see [110] and [111]. Finally, advanced mathematical techniques for handing singular systems via topological degree may be found in [112, 113].

Interval arithmetic can have several purposes in a curve design system. One approach is to specify the control vertices of the B-spline curve as intervals. In this way the constraints of a Lagrange curve design problem can be specified as intervals as well. Then the full power of interval analysis, including non-linear systems solution proving, can be brought to bear in finding all the zeros of a variational problem describing a curve, and proving that certain regions do not have solutions. See [106, 112, 114–117] for the extensive and actively developed capabilities of intervals for general solutions to interval optimization problems. We could propose to extend this for interval B-spline curves, and indeed, some exploratory work has been done to this end. However, we do not utilize this in this thesis. Instead, we will show that even a modest incorporation of basic techniques can have a pronounced effect on the landscape of our optimization problem and the utility of local schemes used to attack it. In particular, extended interval arithmetic, which can narrow the result of an interval computation even when it involves a division by zero [114] will prove quite useful in making progress when singularities are present in the design space.

In this thesis, intervals will be used to describe design space parameters and constraints and also to compute simple interval arithmetic algebraic operations. In the format used, these operations are known as contractors, as they operate by “contracting” the starting intervals by inversion of simple mathematical
relationships between variables.

The basic machinery of interval arithmetic and the techniques of interval analysis (IA) that we use (contractor programming) will be explained in detail in section 8.1. Note that as with automatic differentiation, there is some support for interval arithmetic in Python, such as the package developed and documented by Taschini, [118, 119] and another by Kvesteri, [120], but, particularly with the now nicely maintained package, [118], the code was not as well-developed at the time when the research and interval code development conducted here was ongoing. Therefore, when the interval class was developed here, there was no known implementation which implemented all the needed operators, particularly extended division. The work of [118], however, was of great utility to this project, as we use their exact method to implement the outward rounding needed for correct interval calculations in Python floating-point numbers. At a future time, this project may be re-factored to use [118] as a standard library. However, preliminary investigations have revealed that this may take substantial work in order to ensure that the approach here is both compatible with NumPy and the AD class, as well as workable for the solvers used in interval analysis for nonlinear multidimensional optimization. As such, the current implementation serves as a “test bed” for ideas integrating the power of interval techniques into a parametric CAD/relational constraint program.

In this thesis, the intervals will then be combined with a constraint logic programming (CLP) system. The CLP system will then use those simple interval computations to infer knowledge about the feasible domain. In other words, the system computes the interactions of the constraints and design intervals with one another to contract the total design space to the feasible domain. The full story will have to wait for the technical development in sections 8.1 and 8.2, but we introduce another component, constraint logic programming, in the following subsection.

**Constraint Logic Programming Related Work**

Constraint logic programming (CLP) is a type of higher order programming which sits at the intersection of logic programming, constraint programming, and declarative or rules based systems [121], [122], [123]. Just as with constraint driven form parameter design, the idea is to tell the computer what you want, and the machine uses logic to deduce or infer the best way to go about producing it. The difference between optimization based form parameter design and constraint logic programming is that the underlying algorithms of constraint logic programming are logical at their core, whereas FPD is essentially numerical. Hooker [124] goes into more detail as to the similarities, differences, and profitable merger of, both the optimization (for us,
optimization means FPD in practice) and constraint programming, paradigms. For most researchers, logic programming systems are typically expected to work with symbolic data [123]. Nevertheless, Prolog has been extended to cover constraint logic programming with real numbers [125].

When combined with interval analysis, CLP gains an element of robustness that is missing from other types of logical artificial intelligence systems [7]. The integration of these two paradigms was, it seems, first shown by Cleary, [126] and the main ideas were extended by Older and Vellino [127] in creating BNR-Prolog. Several systems followed, all of which used relational interval arithmetic [128, 129] and often some form of fixed-point solver algorithm in order to process networks of constraints. We will use the arc-consistency algorithm from [121] for this purpose. In order to define arc-consistency, we first define domain consistency.

**Definition 2.0.1.** A variable \( x_i \) is domain consistent if no value of the domain of the node is ruled impossible by any of the constraints.

Next we need the notion of a constraint network.

**Definition 2.0.2.** A constraint network is defined by a graph where we have

- one node for every variable
- one node for every constraint
- undirected edges running between variable nodes and constraint nodes whenever a given variable is involved in a given constraint.

Now let us assume, for the sake of clarity, that there are just two variables, \( x_i \) and \( x_j \), and one constraint \( C \) in our network. For this simple network, arc-consistency is defined as follows,

**Definition 2.0.3.** A given variable \( x_i \) is arc consistent with another variable \( x_j \) if, for every value \( a \) in the domain of \( x_i \), there exists a value \( b \) in the domain of \( x_j \) such that the constraint, \( C \) is satisfied.

There is no reason to restrict our network to one constraint and two variables. The notion of arc-consistency applies across all variables and constraints we choose to introduce into our network.

Hickey [128] notes that more powerful interval constraint satisfaction (and optimization) algorithms, such as global and localized interval Newton’s methods, could be profitably combined with simpler “arc-
consistency” constraint propagation algorithms to achieve the best of both worlds. This is a sentiment echoed by multiple authors. See [124, 130] for details. Constraint propagation complements linear and non-linear solvers. In terms of coding development, we found constraint propagation to be the more elemental or primitive system for constraint solving, best used for setting up feasible problems for the real valued, non-linear solver needed for conventional form parameter design. Digressions aside, the point is that we implement constraint propagation since the more powerful interval optimization algorithms would need it as a supporting tool anyway, since often global or iterative algorithms need to make use of local information in order to make progress.

The workhorse of many logic programming systems, and the underlying algorithm upon which our interval constraint logic system will be based, is an algorithm called “unification,” which will be described in detail later in this paper. This basic algorithm essentially tries to unify one entity with another in a relational equivalence expression (equals), and does so in a manner that checks the types of the entities being unified, handling them appropriately so that data can flow in either direction to make the two variables take on equivalent values, either by deleting inequivalent portions of their corresponding sets, or equivocating their symbols directly if no data has yet been associated to the variables. Thus, unification depends on the data at hand. Since either term in the unification algorithm can be affected by the other, this is known as (binary) relational computation. This together with an extension to handle ternary relations via simple expression inversion, will enable our basic mathematical interval constraints to be relational, in the sense that data is allowed to flow into any of the three components of a mathematical term, from any two of the other terms, suitably combined. This type of constraint logic programming is the intersection of two fundamental technique from “classical AI,” constraint programming, and logic programming, both of which are instances of declarative programming. In this case, we can add the moniker “relational programming” or “relational algebra” as well. Around this central algorithm, using its extension to ternary operators, a relational interval constraint algebra is constructed. This involves relational (and interval valued) algebraic implementations of what would otherwise be standard arithmetic operators. These relational operators act as hull consistency functions in the interval terminology.

In practice, since arithmetic expressions often involve compound terms, a computational tree is built out

---

9Kearfott notes [130] that a plethora of techniques are used on interval optimization problems, and none of them are universally practical.

10The FPD non-linear solver should not be confused with the powerful interval solvers for global optimization!

11Ternary operators in the relational sense are binary operators when acting in the standard procedural way.
of a complete expression, and this tree is walked in reverse, building sub-expressions for each ternary or binary component in a method akin to reverse mode automatic differentiation. This process, though, only need be performed once per rule.

At a higher level, the arc consistency algorithm is used to ensure that when an interval is narrowed by a relational constraint term, the data propagates to any other terms which use that same variable. The propagation of data in this manner is called “constraint propagation [121].” Implementation details can be found in Section 8.2.

We should note that knowledge based systems have been used for basic hull design before. Early design support systems combined expert system reasoning capabilities with empirical rules relating design parameters to performance. Hull modeling facilities were limited to human input of drawings or offsets. The programs could estimate properties of the ship hull based on the crude model data and parametric variation could make use of the estimations of various performance metrics in concert with small changes in the hull offsets to facilitate design space exploration on the machines of that time. Examples of such systems are those of Brower, [131] and Chou [132]. Brower’s system would accept user created lines and searched for feasible designs using performance data interpolated from simple rules and empirical performance graphs. Chou’s [132] system extended the approach to provide a graphical presentation of different design alternatives. Feasibility studies in these systems iterated to find basic design parameters where the sets of performance curves would suggest that such a design would meet customer requirements. Little hull form modeling was possible in these early systems. However, these systems are worth remembering for the efficient estimations of hull performance and performance feasibility intersections which they could compute. It should be noted, however, that the dependence on user input was exacerbated by the use of empirical performance curves as the primary reasoning method. These systems were always limited to modifying baseline designs. Similar systems were constructed by several authors, including [133–135]. Welsh’s INCODE system, [136], was noted by Bertram for having the most comprehensive rules set of its time. This system allowed detailed specification and parametric design of container ships.

More powerful tools began to address the hull form representation, and again made use of the inter-relation of heuristics and empirical relations for design, now together with a basic geometry system. Examples of such systems include [137–140] which perform parametric design with simple user selected shapes. QUAESTOR, by van Hees [136], extended the expert and KB systems that came before it by having a concept of “rejectable” and “non-rejectable” rules. These corresponded to design heuristics and physical laws, respectively.
An expert system called ‘ALFIE’ was notable for utilizing interval arithmetic for expressing uncertain ranges in parameters [141, 142], but the system was focused on ergonomics and human factors, as opposed to ship hull design.

Returning to ship hull design expert systems, Helvacıoglu [143] incorporated cargo capacity and ship speed as parametric rules in his ALDES system. The system also included hierarchical rules for accommodation layout. Other authors, such as Lee in [144] and Aamodt in [145], began to experiment with learning algorithms in concert with expert systems. Typically, such systems were reliant on prior designs from which to extrapolate and build knowledge. Interestingly, Helvacıoglu’s system, along with ALFIE, were two of the few to focus on “geometric based reasoning”, which will play a key role in the system developed here.

Notably, research and development of these systems in the ship hull design industry was never as prolific as in, for instance, the automotive, or even the aerospace industry [5,143,146]. As noted in Helvacıoglu, [143], “The automobile and aircraft industries were targeted for knowledge-based design research with 70 and 170 average research-based articles per year respectively during the 1990s. Meanwhile, only 1.4 research articles per year were observed in the shipbuilding industry (Park & Storch, 2002). This low interest in knowledge-based ship design methods can be attributed to the problems with integration of computer-aided design systems, and insufficient and ineffective geometric reasoning.”

Instead, other applications were found for the rules based reasoning technology. These included collision avoidance, nautical operations, and machinery fault diagnosis within naval architecture and marine engineering. Ship design systems, in contrast, fell into obscurity [5]. Two of the most prominent reasons for this decline with regard to hull design specifically are:

1. “In conceptual design, the task is highly complex and more creative than in other design tasks. As there is still no consensus on the ship design process and how to structure it, automation in reasoning system (sic) seems to be doomed from the outset.” [5]

2. In detailed design, rules are explicitly stated by classification societies and incorporated in design modeling software of the classification societies, e.g. POSEIDON (Germanischer Lloyd), NAUTICUS (Det Norske Veritas), MARS (Bureau Veritas). Additional rules are required in actual detailed design, reflecting for example design for production aspects. These aspects depend on individual shipyards. Larger shipyards incorporate such knowledge via macros and standard part libraries, rather than using KBS systems which are tedious to couple to the predominant commercial design systems.
Additionally, [5] cites slow response time, tedious knowledge extraction from experts, and a decline in academic importance as new techniques become available as reasons for the decline, or rather, lack of uptake, of knowledge based systems in ship hull design.

Another point of view on the dichotomy in research between CAD and expert system ship design is offered by Park and Storch, [146]. They reason that both the complexity of the ship hull CAD design problem, and the uniqueness of each ship design, come into play in discouraging further development of expert systems for especially the geometric hull design of ships. For instance, due to the complexity of CAD design of ship hulls, many firms use more than one CAD design system. In such an environment, yet another interacting system threatens a design firm with promises of additional software integration headaches. Next, the one-off nature of ship hull designs means that it is very difficult to extrapolate general rules that will ensure success from one project to the next. Finally, maintaining a capable expert system is in itself a technological challenge simply due to the rapid pace of change in the digital design era.

But Park and Storch also note that some shortcomings of modern CAD systems are such that expert systems stand ready to address them. Those are as follows:

- CAD systems cannot handle ambiguous or incomplete data
- CAD systems do not provide advice, or other forms of relevant knowledge during the design process.
- CAD systems lack context. They lack data that is not immediately addressable in a CAD system.

But just as CAD systems need expert system aid, the disadvantages of expert systems have prevented their use in that role. Those disadvantages are as follows:

- The integration of the system into existing CAD systems involves substantial problems and difficulties.
- It is not easy to represent ship-design knowledge in computer systems.
- Expert systems are very costly.

**Parametric Tools as Missing Link**

While these disadvantages are substantial, Park and Storch begin to address them by touting the advantages of an integrated database with middle layer for interfacing with particular form parameter tools coupled together with a modular “toolkit” style expert system. This would mitigate the above disadvantages by
• Insulating users from data transfer problems

• Modular, and especially parametric, design approaches enable expert systems to tackle more modest problems through organization of rules.

• An automatic toolkit will significantly reduce time and cost required for developing an expert system.

But in the years since Park and Storch wrote this memo, expert systems have played an increasingly reduced role in ship hull design, while parametric design and robust, set based approaches have become more prominent, at least in research [5]. As Bertram notes, at least in academia, some of this has to do with researchers having exploited the ideas of expert systems as far as they would go, and other reasons were noted before. Traditional expert systems are expensive to create, and hull definition is seemingly not amenable to rules based description.

But the advances that researchers have touted in their stead, especially parametric design, have not caught on either. The reasons are primarily that the parametric tools remain “hard to use” and in turn not as well integrated into design work flows.

What we have come to say is that it is the advances in CAD techniques that have taken place since the 1990s that have paved the way for the reintegration of CAD design with knowledge based reasoning. This is in part because, as Park and Storch said, “Modular parametric design approaches enable expert systems to tackle ‘smaller’ problem by organizing design into a set of rules.” Moreover, here is the key: Parametric tools are difficult to use precisely for issues that expert systems are good at addressing. Namely, feasibility and robustness.

Meanwhile, form parameter tools, and especially machine driven optimization methods have been addressing precisely those design issues that expert systems could not touch. Namely, academia has had run-away success in representing design problems in the parametric format, and even more impressive success in optimization driven by performance, which supplants hard to describe heuristic rule systems whereby experts tried to achieve performance when data was scarce. Performance simulation now means that heuristic data for optimizing performance, which was hard to generate and use in general for ships (especially for new designs), is now supplanted by computerized physical simulation. Further, and just as importantly, form parameter methodologies encapsulate design procedures. These advances are the very same ones that have supplanted research on KBS since the 1990s. For example, in hull modeling there are now powerful generative techniques, as evidenced by more recent publications [9, 10, 61, 63, 71]. To understand these
methods in context, and see how they will aid in a new design of a geometric reasoning system to address shortcomings of the old KB platforms, some additional background on generative methods is needed.

Form parameter design directly addresses Bertram’s point that “In conceptual design the task is highly complex and more creative than in other design tasks. As there is still no consensus on the ship design process and how to structure it, automation in a reasoning system seem to be doomed from the outset”. Indeed, form parameter design has become the leading method for generative design, and for structuring the automated design process. Furthermore, a method which aids in making form parameter design more feasible in practice is the expert or knowledge based, system. The two paradigms strengthen each other. Form parameter systems relieve KBS systems of the following faults (reiterated from above):

- Consensus on the ship design process and how to structure it.
- “Design is often a creative process with few documented rules. FPD provides a design “workflow” so that fewer design rules are needed.
- “Human ‘experts’ have difficulties describing how they proceed in design.” The problem can usually not be decomposed into independent sub-problems.” For simple hull forms, this process is easily described in form parameter design. For more complex hull forms, methods are still being developed and fine-tuned.
- It is not easy to represent ship-design knowledge in computer systems. Again, FPD handles much of this, but is still in active research for complex hull forms.

The complementary benefits do not stop there. The next step is to realize that KBS system have actually been relieved of needing to address the following faults by advances in machine learning and response surface modeling based on objective evaluation utilizing physical simulation of the ship model via computational hydrodynamics, finite element analysis, and other such tools to evaluate the design:

- Representation of domain knowledge for minimization
- Tedious extraction of rules from experts in order to find more optimal designs
- “Different ‘experts’ may also proceed very differently coming to very different designs with no consensus which design is ‘best’” (Now the optimization decides)
- “Design often involves extensive numerical analysis and computer graphics.”

32
The knowledge mapping from design space to performance is generated constructively by the machine learning and response surface tools, together with computational simulation of performance. Optimality is ensured by computation, no longer relying on heuristics. Removal of the human, heuristic element removes subjectivity to some extent. And intensive numerical analysis is no longer a weakness.

And so we see that KBS/expert systems as traditionally used are not enough. They must be augmented with form parameter design models and model building for procedural representation of hull shape. KBS must also be augmented with advanced computational evaluation and search techniques to replace fragile local expert heuristics with bootstrapped knowledge coming from the physics itself.

But this still is lacking. The additional techniques outlined above have their own weak points. Form parameter design has the issue where it is incredibly easy to specify infeasible sets of form parameters. Search with physics optimization is computationally expensive. And so traditional KBS augmented with form parameter tools plus domain learning search tools are still not as efficient and robust as they need to be for industry acceptance of the full automated design paradigm. To fully tackle the robustness and feasibility problem for design in real domains, further ideas from set based and robust design seem to be directly relevant as well, since they provide complementary methods for reducing the search space. Interestingly, ideas from set based design were already a part of generative methods in the past. In the next subsection, we will look at the intersection of these techniques. They will supply KB systems with robustness they traditionally lack. But further, set based tools will confer geometric reasoning power to KBS. These systems will not be knowledge based, but instead constraint based.

In short, neither logical, nor knowledge based, nor expert systems, such as those of Augilar, [139], or van Hees’ Quaestor, [136], have ever been used in concert with form parameter ship hull design techniques like those of Harries, et al., [61].

Here there is no known python implementation where logic programming is augmented to take advantage of interval operations. Several projects have, however, implemented logic programming, for instance, the Python implementation of Datalog, [147], and several Prolog simulations [147], [148], [149].

The implementation given here in section 8.2 is based on the basic unification algorithms detailed by Norvig in [150] and Norvig and Russel in [121], and especially that of [151, 152], with some programming ideas from [153] for the extension to numeric constraints using some ideas from Python’s ast.binop (abstract syntax tree, binary operation) function but altered for relational programming (particularly composing a ternary relational operator from a set of binary relational operators), and then a new implementation of
relational interval operators utilizing the underlying algorithms. The primary architectural resource and inspiration for the logical, constraint based programming system to be developed here is miniKanren [152,154–156]. The system here departs from miniKanren only to expedite a viable solution to the problem at hand, and to incorporate the interval arithmetic directly into the unification algorithm and surrounding constructions, to be given at length in the sections of this thesis.

Summary

In this literature review a key issue inhibiting form parameter design usage growth in industry has been identified as the problem of picking mutually consistent sets of form parameters, especially without reference to or being derived from existing form parameters for a known ship hull. That is, the problem is to use form parameter design outside the realm of modifying known designs. The problem is an important one, as it both inhibits the computational efficiency of ship hull design space search and optimization, but also makes form parameter ship hull design tools harder to use.

Roadmap

The organization of the remainder of this thesis is as follows; first in section 3, the state-of-the-art in computer aided ship hull design (CASHD) will be discussed, including form parameter design methods for ship hull forms. Here the essentials of form parameter design of ship hulls will be stated. In section 4 we recapture the B-spline curve design algorithm as introduced by [52] in order to prepare the way for its implementation by automatic differentiation. A form parameter system for B-spline curves will be developed and then automatic differentiation will be developed, including a sketch of the implementation, in section 5 and an example calculation shown as well. Automatic differentiation and B-spline optimization are then combined in section 6. Next, in section 8.1, interval analysis is developed, including a Python implementation sketch, and in section 8.2, constraint logic programming is explained, including the key unification and arc consistency algorithms. Code examples are given throughout. In section 9, interval analysis and constraint logic programming are combined into an internal domain specific language for constraint consistency. In

12 A quick note on operator arity definitions is in order here. In general, the arity of a function or operation is the number of arguments or operands that the function takes. Relational programming changes things a bit in that the operation takes one additional input over what would be expected in traditional programming or arithmetic — in engineer-speak the “right hand side” has become an input on equal footing with all the rest. Thus a unary assignment operation is a binary relation and a binary operation or action, such as addition, is a ternary relation. This relational nomenclature matches with predicate logic and logic programming [121]. A secondary confusing issue when speaking of implementation issues is that, somewhere “under the hood,” these binary and ternary relations are going to be broken down into unary and binary operations in the standard arithmetic sense.
section 10, the truncated hierarchical B-splines are introduced, and robust methods for computing hierarchical curve solutions to form parameter optimization problems are developed as a simple expansion on the standard methods for Lagrangian curve solving developed earlier. In section 11, this framework is used to filter out infeasible form parameter design subspaces and a demonstration project is shown to generate viable hull form geometry similar to that of an offshore supply vessel (OSV). Here we employ a random number generator to choose form parameters from the parameterized feasible domain. A set of basic naval architecture rules are used to ensure consistency of form parameters across aspects of the hull design. The truncated hierarchical basis is used in the final step to ensure that all constraints are met exactly. Results and example hulls are generated above and examined in section 11. Discussion, summary, and conclusions are given after this subsection 13 and potential next steps given in section 13.1.

A few words on notation are in order: vectors, e.g. a B-spline vertex $\underline{V_i}$, are marked with underlined variable names. The dot product between two vectors is written as a matrix multiplication $a^T b = \sum_i a_i b_i$. The superscript $T$ indicates transposition, i.e. exchange of rows and columns. Double underscores denote matrices.

When we come to interval analysis, notation will also follow that of Kearfott [109], with modifications to follow the standard practices of The Journal of Reliable Computing, [157] as mentioned above. In the interval section of this paper, boldface will denote intervals, lower case will denote scalar quantities and upper case will denote vectors and matrices. Brackets “[◦]” will delimit intervals while parentheses “(◦)” will delimit vectors and matrices. Underscores will denote lower bounds of intervals and overscores will denote upper bounds of intervals. Corresponding lower case letters will denote components of vectors. The set of real intervals will be denoted by $\mathbb{IR}$. Further details will be given in the sections where they are needed.

---

13 Each form parameter is a real interval parameterized on the domain $[0., 1.]$
Chapter 3

Form Parameter Design of Ship Hull Geometry

Introduction and Greater Context

Form parameter design is the process of developing a ship hull form by solving a set of optimization problems. Each optimization problem solves for a particular curve. These problems consist of constraints, which are the form parameters, and some fairness measure, usually thought of as an energy form. The particular details of this curve design task will wait until section 4. In this section, we give a broader view. Here we describe how the entire hull is generated using many such curve solutions.

The typical practice, and the one followed here, is to develop a lines plan starting from a small set of basic curves. These curves are chosen such that all other ship lines can be deduced from this set and the essential shape characteristics are determined and controlled by this set. In our system, these curves are usually fixed once they are generated. In our system these are the sectional area curve (SAC), waterline curve and center-plane profile curve.

A key difference between this system and typical form parameter systems is that in this system, the user need only partially specify the design space. The user may specify greater detail if desired, but the system is capable of filling in any needed information in the design, and indeed does so automatically, and logically. Furthermore, the system is capable of automatically selecting sets of consistent form parameters from within the design space itself. These capabilities will be explained in further sections. In this section, we will ignore where the form parameters come from, and instead concentrate on how they are used in order to generate the hull geometry.

Table 3.1 displays the inputs available for setting principle dimensions and hull form relationships of

\[\text{Table 3.1 displays the inputs available for setting principle dimensions and hull form relationships of} \]

\[\text{1 The lines plan consists of waterlines, sections and buttock lines, lying in orthogonal planes parallel to the coordinate axis.} \]
Principle Dimensions, Coefficients, and Other Inputs

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{WL}$</td>
<td>Length of the Waterline</td>
</tr>
<tr>
<td>B</td>
<td>beam</td>
</tr>
<tr>
<td>D</td>
<td>draft</td>
</tr>
<tr>
<td>$\nabla$</td>
<td>volume of displacement</td>
</tr>
<tr>
<td>$C_B$</td>
<td>block coefficient</td>
</tr>
<tr>
<td>$C_P$</td>
<td>prismatic coefficient</td>
</tr>
<tr>
<td>$A_M$</td>
<td>midship section area</td>
</tr>
<tr>
<td>$C_M$</td>
<td>midship coefficient</td>
</tr>
<tr>
<td>$L_{CB}$</td>
<td>longitudinal center of buoyancy</td>
</tr>
<tr>
<td>$C_{LB}$</td>
<td>length to beam ratio</td>
</tr>
<tr>
<td>$D_{LR}$</td>
<td>displacement to length ratio</td>
</tr>
<tr>
<td>$L_{FSAC}$</td>
<td>length of the flat of sectional area curve</td>
</tr>
<tr>
<td>$A_{WP}$</td>
<td>waterline area</td>
</tr>
<tr>
<td>$C_{WP}$</td>
<td>waterline coefficient</td>
</tr>
<tr>
<td>$L_{FWL}$</td>
<td>length of the flat of waterline curve</td>
</tr>
<tr>
<td>$C_{CP}$</td>
<td>center profile area</td>
</tr>
<tr>
<td>$A_{CP}$</td>
<td>center profile area coefficient</td>
</tr>
<tr>
<td>$L_{FCPK}$</td>
<td>length of the flat of center plane keel profile curve</td>
</tr>
<tr>
<td>$D_t$</td>
<td>Transom drop (from waterline to centerplane)</td>
</tr>
<tr>
<td>$C_{ffAMbulb}$</td>
<td>Bulbous bow to Forward Fairness Area Coefficient</td>
</tr>
<tr>
<td>$C_{ffzLbulb}$</td>
<td>Bulbous bow length to forward fairness station location Coefficient</td>
</tr>
</tbody>
</table>

Table 3.1: Principal Inputs for the Bare Hull FPD System

the bare hull form at the highest level. Most of these effect the SAC curve, some will affect the waterline and keel profile, and many of them act as constraints between two or more curves. Altogether, the inputs on this table are more than sufficient for determining the broad shape of the vessel. Again and to emphasize the greater context, these terms and their requisite definitions are actually rules. They are relational expressions which constrain each other. This will be explained in section 8.3.1. Here, we may merely take them as inputs to the form parameter design system. Some inputs will appear in multiple rules. The highly interconnected rules network will ensure consistency of the form parameters with each other, and also goes some way to highlight the problem that users of other form parameter systems must solve implicitly — that of finding self-consistent combinations of input form parameters. The initial inputs are given in Table 3.1.
Procedures

With the initial inputs specified, curve generation proceeds as it generally does in any form parameter system. Accordingly, what follows is similar to the scheme of [52]. Extensions to this method to allow for complex hull form topologies will come from constraint programming on one hand, and extensions to the B-spline representation to allow for local refinement on surfaces, on the other.

Note that for this demonstration system, a fairly coarse description of the sectional area curve is made. This system is fully automated, as such it is imperative to successful development to add complexity slowly, ensuring that the system never loses its properties of robustness. With the success of this demonstrator program, the path is clear to extend the constraint relationships to account for more complex interactions. For now, we take as input fractional section area properties of the bulbous bow which fixes the sectional area at the bow of the ship, and we take similar input for the aft-most section. Furthermore, we assume the bulbous bow terminates at the nose of the bare hull form. This simplified the development process so that we could focus on other areas of shape control.

We now proceed to explain how form parameter design proceeds from the high level description to more detailed levels and finally to the full hull B-spline surface representation.
Hull Form Generation

The basic map of the hull form generation process is shown in Figure 3.1. Note for our particular system, the bare hull coordinate system is right-handed, with the \( z \) coordinate running positive aft along the longitudinal axis, the \( x \) coordinate positive to starboard, and the \( y \) coordinate positive vertical, and the origin at the bow, keel, centerline of the hull. For display purposes later, we rotate the hull for easier manipulation in Rhino. As shown in the chart referenced above, the basic form parameter ship hull design system is then assembled in a four step process as follows:

1. Curves of Form: Using the curve solver developed in section 4, the new form parameter system accepts a set of form parameters which are then solved to produce a set of basic form curves.

2. Transverse Curves: The transverse sectional curve properties are then derived from the curves of form, and solved, again using the same curve solver.
   - For instance, at every section where a curve is to be generated, that curve will take on a sectional area given by the height of the sectional area curve at the same station.
   - Other form parameters for sections are similarly derived. Stating and ending locations come from the design waterline and center plane keel profile curve.

3. Longitudinal Space Curves. Then a set of longitudinal 3D, or “space” curves are generated, again using the optimization procedure. This time the constraints are that the curves must pass through (exactly interpolate) the vertices of the transverse curves.
   - Thus the number of longitudinal curves equal the number of transverse vertices on one curve.
   - And in turn the number of transverse curves governs the minimum number of longitudinal vertices needed, to exactly interpolate the transverse vertices.

4. A skinning algorithm, due to Woodward [158], is then used to generate a B-spline surface which exactly interpolates these section curves, generating a basic bare hull shape.
   - This is accomplished by generating a set of longitudinal curves which exactly interpolate the control vertices of the transverse curves.
   - The vertices of the longitudinal curves then become the control vertex network of the new surface.
This is a simple linear process, but if desired, the skinning algorithm can be augmented to again use the curve solver detailed above to enforce additional constraints, or to fair the skinning curves.

**Curves of Form**

A simple set of curves of form are used in this thesis, as follows:

1. Sectional Area Curve (SAC)

2. Design Waterline Curve (DWL)

3. Center-plane Keel Curve (CPK)
Sectional Area Curve

The sectional area curve (SAC, or “SA curve” to aid in the exposition below) is of utmost importance to any hull design, as it controls the distribution of volume over the length of the ship. In form parameter design, the SA curve informs the transverse curve generation procedure of what the (half) area parameter will be for a transverse curve at any station along the length of the hull. Our sectional area curve is no different in this regard. See, e.g. Harries [52] for details. In addition to this role, our sectional area curve is designed for a bulbous from the very start in this formulation. For this reason, the starting vector for FPD given in table 3.1 accepts data about the bulbous bow. The full procedure relies on the constraint solver to be specified later in section 11.2, but we may still give the relevant details for integrated SAC design now.

Briefly, before FPD begins, during the initial constraint solver computations, which pick out a feasible design from the design space, feasible design vectors for both the bare hull and bulbous bow are generated. This can be done since the particulars of the bulbous bow depend on the form parameters already determined at the “constraint solve time.” Therefore, we know the cross-sectional area of the final bulb before any curves have even been solved for. Accordingly, we may build this into the SAC curve here.

Determination of Sectional Area Curve “Starting Height”

The procedure is to use the known bulbous bow cross-sectional area and length, again, both determined by constraint solver, to determine what portion of the SA curve here to apportion for the bulbous bow SA curve. Crucially, the constraint solver also accepts as input some fraction of the “transverse forward fairness curve” should go towards fairing in the bulbous bow. Therefore, if we assume a starting position of the bulbous bow, already know its length and cross-sectional area (already solved to be compatible with the bare hull form parameter inputs), then we can say what the termination height of the SA curve should be. We assign the forward end point of the SAC a height which is equal to the sectional area of the bulb at that point, since there, at the very tip of the nose of the “bare hull” surface, all the volume remaining must be accounted for within the bulbous bow. Hence the bulbous bow section area determines the forward starting “height.”

Natural Determination of Bulbous Bow Sectional Area Curve

Furthermore, once a SAC is designed, we will also know the area to be used at the forward fairness section curve. With this we can construct a bulbous bow sectional area curve, including fairing into the main hull at
Figure 3.3: Normalized Sectional Area Curve Showing flat mid-section and fore and aft transverse area accommodations for the eventual bulbous bow and transom stern. Longitudinal center of gravity is also fixed in this curve, though not shown here. See section 7.1 for more details on SAC optimization. An overview of the form parameters used for this particular curve, and thus those used in the eventual hull forms developed here, is given below in table 3.2.
Figure 3.4: Normalized Sectional Area Curve Also showing concurrently designed Bulbous Bow Sectional Area Curve and fairing area (small curve to bottom left, where the flat portion is actual bulbous bow and the angled portion handles fairing from the bulbous bow to the hull proper). Furthermore, the bulbous bow itself is the flat section of the smaller curve, while the inclined section shows that the fairing of the bulb into the hull will show an increasing sectional area moving aft. Note that fairing curve can be influenced by rule. This will be important later in the thesis for “nicely” integrating the complex hull.
### Sectional Area Curve Form Parameter Inputs

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>SAC area (Hull volumetric displacement, $\nabla$)</td>
</tr>
<tr>
<td>$A_M$</td>
<td>SAC max height at midship (midship section area)</td>
</tr>
<tr>
<td>$x_c$</td>
<td>SAC LCB (vessel longitudinal center of buoyancy, LCB)</td>
</tr>
<tr>
<td>$L_{FSAC}$</td>
<td>length of the flat of sectional area curve</td>
</tr>
<tr>
<td>$\alpha_1$</td>
<td>Tangent angle at the start of the flat section</td>
</tr>
<tr>
<td>$C_1$</td>
<td>Curvature at the start of the flat section</td>
</tr>
<tr>
<td>$\alpha_2$</td>
<td>Tangent angle at the end of the flat section amidships</td>
</tr>
<tr>
<td>$C_2$</td>
<td>Curvature at the end of the flat section</td>
</tr>
<tr>
<td>$V_n$</td>
<td>Transom drop, $D_{\text{bulb}}$ (from waterline to center-plane)</td>
</tr>
<tr>
<td>$V_0$</td>
<td>Bulbous bow aft cross-sectional area, $A_{\text{Mbulb}}$ and also cross-sectional area at the most forward station on the hull.</td>
</tr>
<tr>
<td>$L_{\text{bulb}}$</td>
<td>Bulbous bow length (straight bulbous bow sections)</td>
</tr>
</tbody>
</table>

Table 3.2: SAC Form Parameters

To summarize the discussion leading to this subsection, the constraint solvers for bare hull and bulbous bow can determine a great many form parameters and inter-relationships for us in advance, including basic bulbous bow properties like cross-sectional area, and length of the bulbous bow. We exploit this in order to build an integrated hull SAC and then bulbous bow SAC including fairing area at the outset of the hull design process.

The form parameters used in the FPD solution of the hull SAC are given in table 3.2. The resulting sectional area curve is shown in figure 3.3. All form parameters of the sectional are curve of the hull are solved as equality constraints. In general, all form parameters in this thesis are equality constraints, except where noted. Also of note, the SA curve was solved as a single segment. The implementation of the constraints to make this possible was simplified by the extension of the form parameter design optimization problem to utilize automatic differentiation. This is described in section 6.

Now that we have described the form parameter design problem setup for the sectional area curve, we move on to the other curves of form. Solutions will wait until the general variational curve problem is developed in detail in section 4 and examples of solved curves are given in section 7. Here, we next describe the problem setup for the design waterline curve.
A design waterline curve requires certain parameters to be defined, as listed in the Table 3.3: DWL Form Parameters. The table outlines the following parameters:

- $A$: Waterline area, $A_{WP}$
- $x_C$: Longitudinal center of flotation, $X_{LCF}$ - to be least square approximated
- $L_{F_{SAC}}$: Length of the flat of the DWL
- $\alpha_B$: Tangent angle at the bow
- $y_E$: Waterline max width, $B$ (half beam of waterline)
- $\alpha_1$: Tangent angle at the start of the flat section
- $C_1$: Curvature at the start of the flat section
- $\alpha_E$: Tangent angle at the transom
- $C_E$: Curvature at the transom
- $B_{transom}$: Half-width at transom

**Table 3.3: DWL Form Parameters**

**Design Waterline Curve**

Note that the term “design waterline,” means that this is a designed waterline curve, but this is not the line at which ship will float at design draft. Instead this will be more of like a deck definition curve in terms of vertical positioning. Since we are merely demonstrating the capabilities of the program, any waterline curve will suffice. Furthermore, our designed waterline is an important parameter in this solution, as it determines the outermost breadth of our transverse sections. If this curve is taken as a deck curve then we could add a true design waterline curve using similar constraints. The two waterlines would be connected by transverse curves at the cost of an additional interpolation constraint, and possibly tangent and curvature constraints as needed. For more details and usage of the design waterline curve see Nowacki [9]. Our DWL deck curve is also solved using FPD as similar to Harries [52]. One particular difference between waterlines in [9, 52] and the waterline developed here is that we incorporate a transom hulled vessel. For this transom we include an extra form parameter to accommodate the aft breadth of the ship. The form parameters of the design waterline are given in table 3.3.

Finally, the DWL curve was solved as a single segment. As usual in this thesis, a generic curve solver implemented via automatic differentiation was used to solve for this curve. See table 3.3 for the form parameters used in generating the DWL curve.

---

2 Indeed by using a higher waterline curve, we increase the topological difficulty of mating the bulbous bow to the bare hull form. This is the true purpose of designing a deck curve instead of a design waterline — to form one contiguous hull surface, and show that we can mate the bulbous bow to it.
Figure 3.5: Normalized Designed Waterline Curve.
Center Plane Keel Curve Form Parameter Inputs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>Profile area, $A_{CP}$ (solved as equality constraint but later modified by bulbous bow)</td>
</tr>
<tr>
<td>$x_C$</td>
<td>Longitudinal center of profile area. (Least square approximated)</td>
</tr>
<tr>
<td>$L_{F_{SAC}}$</td>
<td>Length of the flat of the CPK</td>
</tr>
<tr>
<td>$D$</td>
<td>CPK max draft</td>
</tr>
<tr>
<td>$C_B$</td>
<td>Curvature at the start of the bow (Least square approximated)</td>
</tr>
<tr>
<td>$\alpha_1$</td>
<td>Tangent angle at the start of the flat section</td>
</tr>
<tr>
<td>$C_1$</td>
<td>Curvature at the start of the flat section</td>
</tr>
<tr>
<td>$\alpha_2$</td>
<td>Tangent angle at the end of the flat section</td>
</tr>
<tr>
<td>$C_2$</td>
<td>Curvature at the end of the flat section</td>
</tr>
<tr>
<td>$\alpha_E$</td>
<td>Tangent angle at the transom</td>
</tr>
<tr>
<td>$y_E$</td>
<td>Transom drop (from waterline to center-plane)</td>
</tr>
</tbody>
</table>

Table 3.4: CPK Form Parameters

Center-plane Keel Profile Curve

The center-plane keel profile curve is an important curve influencing the flow of water to the stern and dictating the shape of the bow profile of the vessel. Our CPK curve form parameter design follows that of Harries [52]. The exception here is that we allow again for the transom stern. Form parameters of the center plane keel are given in table 3.4.

As before with SAC and the DWL curve, the CPK curve was solved as a single segment. The implementation of the constraints to make this possible was again simplified by using automatic differentiation. See table 3.4 for the form parameters used in generating the CPK curve.

Note that the CPK curve is solved so that it’s longitudinal center of area will be pulled forward in anticipation of the addition of the bulbous bow to the bare hull. This facilitates construction of the complex hull, but does not ensure that bare hull and complex hull have exactly the same center plane profile area. Future work could ensure this area is constant throughout the design process.
Figure 3.6: Normalized Center-plane Keel Profile Curve. Forward sweep developed (at bow, left) in anticipation of the addition of the bulbous bow. Transom “drop” aft is controlled by $y_E$. 
Finally, all curves of form are normalized to 1 and plotted in table 3.7. With the completion of form parameter design of the curves of form, we are ready to move on to the next phase of form parameter design.

**Transverse Curves**

Next we solve for a set of transverse hull curves. In practice, following Harries [52], we define our transverse B-spline curves to have 7 vertices each. This allows sufficient freedom to meet the basic form parameters needed, as described in section 4. Boundary conditions, which are the form parameters, determine the transverse shape of the keel and the angle and perhaps curvature where the transverse meets its design waterline. The number of transverse curves can vary, depending on how the final surfaces are to be built (a single surface with multiple zones might need more transverse curves as guides, where boundary conditions might have sufficed before). In general, we use the following transverse curves, labeled as in figure 3.8:

1. **Bow curve.** This can be a simple stem curve or, more likely be composed of the following two components:
   - Stem from the highest point at the waterline or deck curve to the lowest point at the upper edge of the bulbous bow
   - Bulbous bow forward outline.

2. **Forward fairness curve.**
   - This curve is very important in defining the forward volumetric distribution or “shape” of the hull.
   - More will be said about this curve later, but the lower portion includes volume that aids the (longitudinal) lines in sweeping into the bulbous bow.
   - The upper portion aids the lines in sweeping cleanly to the stem.
   - An appropriate distribution between upper and lower portions must be sought.

3. **Forward mid-ship transition curve.**
   - This curve ensures a smooth transition to the flat midship surface.

4. & 5. **Mid ship transverse curves.**
   - These curves define the flat mid-ship section and it’s bilge radius.
   - Therefore there is at minimum, one curve at the forward edge of the midship section, and one curve at the aft edge of the flat midship section.
   - They may have some sweep forward and aft.

6. **Aft mid-ship transition curve.**
   - This curve ensures a smooth transition from the flat midship surface to the run of the stern.

7. **Aft fairness curve.**
   - This curve controls the run to the transom.
Figure 3.7: Normalized primary SAC, bulbous bow and blending section SAC, and a designed waterline and center plane keel profile curve. The bulbous bow SAC is the short curve visible at bottom left. The fairing area is shown as well, though hard to discern here. The full length curves at bottom are the design waterline and center plane keel. Note that the design waterline (half breadth) and center plane keel have similar magnitude of maximum depth and magnitude of maximum width on this vessel. We might use a draft to beam rule if we wish to control this.
Figure 3.8: Simple transverse curves network. This view shows the longitudinal positioning of these curves. Bow curve at far right. Transom curve at far left. An initial bow curve is shown. It will be replaced when the form parameter designed bulbous bow is added.

- Together with the aft mid-ship transition curve, they control the flow of the water to the transom, and over the propellers in the final design.

8. Transom curve.

- This curve defines the transom.
- Typically there is some drop in height from the transom curve at its fullest breadth to its position at the centerline.

A typical set of automatically generated transverse curves is diagrammed by longitudinal station location in figure 3.8. Cross sections of the same curves are shown in figure 3.9.

**Longitudinal Curves**

Longitudinal curves are designed to interpolate the control vertices of the transverse B-spline curves. This follows the hull form skinning process of Woodward [158] as used by Harries [52] in form parameter design to create surfaces which exactly match the original transverse B-spline curves.

These requirements are translated into point constraints for the individual longitudinal curves. The automatic differentiation framework will handle these constraints just the same as any other. See sections 4 and 6 for details.
Figure 3.9: Transverse section curves.
Figure 3.10: Simple (bare hull) Longitudinal Curves. This is what a set of bare hull curves would look like without modifications to support bulbous bow incorporation.

Figure 3.11: Simple (bare hull) Longitudinal Curves. Note that the form parameter designed bulbous bow has not been added at this stage, but the center-plane keel profile curve has been swept forward in anticipation of the new bulb. Future work could ensure that center plane area and distribution of area remained constant when comparing bare hull to complex hull.
Note that we could have used linear interpolation for this purpose, as described in Piegl and Tiller [24]. In practice though, more flexibility was achieved with the full nonlinear solver than with a simple linear inversion to solve for the longitudinal control points.

In the end, though, we chose to subdivide the hull form and continued using nonlinear optimization for the longitudinal curves and now used that flexibility to experiment with boundary conditions for the longitudinal curves. The full details are given in the complex hull section 11.2. It’s well worth emphasizing, however, is the critical usefulness of being able to experiment with boundary conditions. This is afforded by the nonlinear solver with ease, since these are simply added or removed from the problem functional using automatic differentiation.

A basic set of curves that might look something like that shown in figure 3.10 though here we show that we will be changing the center-plane keel profile to incorporate the bulbous bow. (Transverse B-splines will not change, except for the bow curve.) The shifted center plane keel curve is shown in the longitudinal curves given in figure 3.11.

**Summary**

So here we have seen the main details of a form parameter ship hull design problem emerge. Namely, the FPD of a ship is performed by solving a sequence of planar curve optimization problems, followed by a set of three-dimensional or, “space curve” optimizations of the longitudinal curves where the transverse vertices become the primary constraints of the space curve optimization. Furthermore, the automatic differentiation formulation of the problem, to be specified in sections 5 and 6, is critical to allowing for experiment with boundary conditions, so as to find optimal intra-surface boundary conditions, as will be detailed in 11.2 and especially 11.3.

Before we can get to those details, we must develop the solver for B-spline curves. As we have shown here, in this thesis, all curves are generated in the B-spline representation, or, as we shall see later on, hierarchical extensions of the same. Accordingly, in the next section, we shall examine in detail how to set up and solve a general curve design and optimization problem for B-spline curves.
Chapter 4

Lagrangian Curve Design of B-splines

NURBS and B-splines form the backbone of modern 3D modeling systems. A B-spline of order \( k \) is defined as the linear combination of \( (n + 1) \) vertices \( \mathbf{V}_i \), weighted with the values of the B-spline basis functions \( N_{i,k}(t) \) (with \( i = 0, 1, 2, \ldots, n \)).

\[
\mathbf{q}(t) = \sum_{i=0}^{n} \mathbf{V}_i N_{i,k}(t) \quad (4.1)
\]

The B-spline basis functions are recursively defined by a scheme first presented by de Boor and Cox [21, 23]. Required input is the order \( k \) and a so-called knot vector, for details see [24]. In our application, only uniform knot vectors are used because they only depend on the order and the number of control vertices. Importantly, basis functions are independent of the vertex coordinates.

The coordinates of the curve point \( \mathbf{q} \) may be rewritten as a dot product of vertex coordinates and basis functions. For a planar curve we get:

\[
\mathbf{q}(t) = \left( \begin{array}{c} q_x(t) \\ q_y(t) \end{array} \right) = \left( \begin{array}{c} \sum_{i=0}^{n} x_i N_{i,k}(t) \\ \sum_{i=0}^{n} y_i N_{i,k}(t) \end{array} \right) = \left( \begin{array}{c} \mathbf{v}_x^T N_k(t) \\ \mathbf{v}_y^T N_k(t) \end{array} \right) \quad (4.2)
\]

The \( x \)- and \( y \)-coordinates of vertices have been collected into the vectors \( \mathbf{v}_x \) and \( \mathbf{v}_y \) respectively. The basis functions values \( N_{i,k}(t) \) form the vector \( \mathbf{N}_k(t) \).

\[
\mathbf{v}_x^T = (x_0, x_1, x_2, \ldots, x_n)
\]

\[
\mathbf{v}_y^T = (y_0, y_1, y_2, \ldots, y_n) \quad (4.3)
\]

\[
\mathbf{N}_k(t)^T = (N_{0,k}(t), N_{1,k}(t), N_{2,k}(t), \ldots, N_{n,k}(t))
\]
This arrangement simplifies the notation of subsequent equations and the actual evaluation of form parameters.

Harries first introduced the form parameter driven design of B-spline curves into the hull design of ships [52, 61]. We recapture his approach here in order to explain how automated differentiation simplifies its implementation and extends its flexibility.

Harries [52] formulated the process of constructing fair B-spline curves that satisfy a set of design constraints as a Lagrangian optimization (minimization) problem. Form parameters serve as equality and possibly inequality constraints. The set of form parameters could be made up of the end points, tangent angle and curvature at the end points, area between the curve and the $x$-coordinate axis and the centroid location, or anything else we might conceive of as a property germane to the curve. A typical subset of form parameters is shown in Figure 4.1. A fairness functional is used as measure of merit for the optimization. The vertex coordinates form the vector of free variables.

Provided that the number of free variables is greater than the number of form parameters, the problem is stated as follows:
B-spline curve design applying fairness criteria:

Find a two dimensional open B-Spline curve

\[ q(t) = \sum_{i=0}^{n} V_i N_{i,k}(t) \]  \hspace{1cm} (4.4)

of given order \( k \), number of vertices \( n + 1 \) and specified knot vector \( T \) such that the user’s chosen subset of selected form parameters is met and, in addition, the curve is considered good in terms of a problem-oriented fairness measure.

We first discuss the connections between form parameters, constraints, and vertex coordinates.

**Constraints**

The basic curve design problem has only equality constraints derived from the form parameters (Figure 4.1):

1. End point coordinates

\[
\begin{align*}
  x_B &= q_x(t = t_B) \quad \text{beginning point } V_0 \\
y_B &= q_y(t = t_B) \\
x_E &= q_x(t = t_E) \quad \text{end point } V_n \\
y_E &= q_y(t = t_E)
\end{align*}
\]  \hspace{1cm} (4.5)

2. Beginning and end tangency angles \( \alpha_B \) and \( \alpha_E \) which are defined by the angle between the first leg of the control polygon and the x-axis and the last leg of the control polygon and the x-axis, respectively. In general, a tangent angle may be defined anywhere along the curve.

3. Curvatures at the beginning and end

\[
\begin{align*}
c_B &= \frac{\ddot{q}_x \tilde{q}_y - \ddot{q}_x \tilde{q}_y}{\sqrt{\dddot{q}_x^2 + \dddot{q}_y^2}} \bigg|_{(t=t_B)} \\
c_E &= \frac{\ddot{q}_x \tilde{q}_y - \ddot{q}_x \tilde{q}_y}{\sqrt{\dddot{q}_x^2 + \dddot{q}_y^2}} \bigg|_{(t=t_E)}
\end{align*}
\]  \hspace{1cm} (4.7)  \hspace{1cm} (4.8)

57
First and second order derivatives of the curve point coordinates with respect to the curve parameter \( t \) are denoted with \( \dot{q}_{x,y} = \frac{dq_{x,y}}{dt} \) and \( \ddot{q}_{x,y} = \frac{d^2q_{x,y}}{dt^2} \) respectively.

4. The area between the curve and \( x \)-axis is derived with the help of Stokes’ integral theorem.

\[
\iint_A dA = \frac{1}{2} \left[ \oint_A x \, dy - \oint_A y \, dx \right]
\]

Applying this to the area under the B-spline curve yields

\[
A = \frac{1}{2} \left[ \int_{\beta}^{\tau} (q_y \dot{q}_x - q_x \dot{q}_y) \, dt + x_{EY} - x_{BY} \right] \quad (4.9)
\]

5. The area between curve and \( y \)-axis is given by

\[
A_y = \frac{1}{2} \left[ \int_{\beta}^{\tau} (q_y \dot{q}_x - q_x \dot{q}_y) \, dt + x_{BY} - x_{EY} \right]
\]

6. Longitudinal and lateral centroid coordinates \( x_c = M_y/A \) and \( y_c = M_x/A \) of area \( A \). With the first order moments provided by

\[
M_y = x_c \, A = \frac{1}{3} \left[ \int_{\beta}^{\tau} q_x (q_y \dot{q}_x - q_x \dot{q}_y) \, dt + x_{EY} - x_{BY} \right] \quad (4.10)
\]

\[
M_x = y_c \, A = \frac{1}{6} \left[ 2 \int_{\beta}^{\tau} q_y (q_y \dot{q}_x - q_x \dot{q}_y) \, dt + x_{EY}^2 - x_{BY}^2 \right] \quad (4.11)
\]

The area enclosed by two curves is computed as the area of the lower curve with the \( x \)-axis subtracted from the area of the upper curve with the \( x \)-axis.

\[
A = A_{\text{upper}} - A_{\text{lower}} \quad (4.12)
\]

Furthermore, the area enclosed by a set of \( J \) B-spline curves is given as a sum of the contribution to the area
made by each B-spline curve $j$:

$$A = \sum_{j=1}^{J} A_j \quad (4.13)$$

with

$$A_j = \frac{1}{2} \int_{t_B}^{t_E} \left[ q_x \frac{dq_y}{dt} - q_y \frac{dq_x}{dt} \right] dt \quad (4.14)$$

Form parameters involving integrals pose an additional difficulty because the control vertex coordinates (free variables of the optimization) appear as part of the integrand. Changes in the free variables require re-evaluation of the integrals and make the computation of the derivatives of the Lagrangian function more cumbersome. The same is true for the contributions to the fairness functional (4.20) discussed in the following section. It is therefore advantageous to extract the vertex coordinates from the integrals. As noted by [52], most of the equations can be rewritten using the vector definitions from Equation (4.3). For example, Equation (4.9) transforms into

$$2A = v_T \int_{t_B}^{t_E} \left[ \left( M(t) - M^T(t) \right) dt \right] v_x + x_{EYE} - x_{BYY} \quad (4.15)$$

Similar equations may be derived for other form parameters. The coefficients of the square matrix $M(t) = \{M_{ij}(t)\}$ are products of B-spline basis functions with basis function derivatives and independent of the vertex coordinates. The matrix is defined as the outer product of the vectors $N_k$ and $\dot{N}_k$.

$$M = N_k \frac{dN_k^T}{dt} \quad (4.16)$$

with

$$M_{ij}(t) = N_{i,k}(t) \frac{dN_{j,k}(t)}{dt} \quad \text{with } i, j = 0, 1, 2, \ldots, n \quad (4.17)$$

The integrals involving the matrix $M$ are computed only once and stored before optimization begins.

Note that computation of any derivatives of the basis functions, i.e. terms such as $\frac{dN_{i,k}(t)}{dt}$, proceeds via
de Boor spline differentiation [24]. The major computational task in the evaluation of area and centroid coordinates is the evaluation of B-spline basis functions and their derivatives.

The user can select any subset of these parameters as constraints for the optimization. Equality constraints are implemented through the Lagrange multiplier technique. Inequality constraints are first converted into equality constraints by the use of slack variables. As described next, this recasts the constrained optimization problem into an unconstrained multivariate problem.

**Objective Function**

During the optimization process curves are compared based on a measure of merit for fairness. For an overview of fairness criteria and their application see [159]. We employ pseudo-norms $E_K$ which approximate squared arc length, bending, and curvature energy in the curve.

$$E_K = \int_{t_B}^{t_E} \left[ \left( \frac{d^K q_x}{dt^K} \right)^2 + \left( \frac{d^K q_y}{dt^K} \right)^2 \right] dt \quad \text{with} \quad K = 1, 2, 3$$

(4.18)

The total arc length may also be included directly in the objective function which helps to suppress self-intersecting loops in the faired curve. The arc length is computed as

$$S = \int_{t_B}^{t_E} \sqrt{\left( \frac{dq_x}{dt} \right)^2 + \left( \frac{dq_y}{dt} \right)^2} dt$$

(4.19)

$E_K$-pseudo-norms and arc length $S$ are combined into a weighted objective function $f$ which we subsequently call the fairness functional:

$$f = e_1 E_1 + e_2 E_2 + e_3 E_3 + sS$$

(4.20)

The weights $e_K$ with $K = 1, 2, 3$ and $s$ are used to switch $E_K$-pseudo-norms on or off. Based on the fairness functional (4.20) and a selection of equality constraints, the formal optimization task (4.4) is cast in more mathematical terms.

*Compute the unknown vertices $V_i$ with $i \in [0, n]$ of the B-spline curve $q(t)$ by minimizing the objective function*
\[ f = e_1 E_1 + e_2 E_2 + e_3 E_3 + s S \]

subject to any or all of the following equality constraints:

\[ \begin{align*}
    h_1 &= 0 = q_x (t = t_B) - x_B \\
    h_2 &= 0 = q_y (t = t_B) - y_B \\
    h_3 &= 0 = q_x (t = t_E) - x_E \\
    h_4 &= 0 = q_y (t = t_E) - y_E \\
    h_5 &= 0 = \alpha_{B_{\text{actual}}} - \alpha_B \\
    h_6 &= 0 = \alpha_{E_{\text{actual}}} - \alpha_E \\
    h_7 &= 0 = c_{B_{\text{actual}}} - c_B \\
    h_8 &= 0 = c_{E_{\text{actual}}} - c_E \\
    h_9 &= 0 = A_{\text{actual}} - A \\
    h_{10} &= 0 = M_{y_{\text{actual}}} - x_c \cdot A \\
    h_{11} &= 0 = M_{x_{\text{actual}}} - y_c \cdot A
\]

(4.21)

At this point, the set of \( m \) equality constraints are combined with the objective function, \( f \), to form the augmented fairness, or Lagrangian, functional \( F \). Employing \( \lambda \) for the Lagrange multipliers \( \lambda_j \neq 0 \), the Lagrangian functional is

\[ F = f + \sum_{j=1}^{m} \lambda_j h_j \]  

(4.22)

If, as is often the case, some of the constraints can be used to pre-specify some of the edge vertices, the system of equations could be reduced, by for example setting the "y" value of the second and second to last control points in order to pre-set the tangent angles at both ends of the curve. See [52] for details.

However, our method allows tangents to be specified anywhere. Additionally, point constraints can be extended such that the curve interpolates any point in general. The same is actually true for curvature. In addition, constraints may define location of cusps, or specific shapes that must fit into the interior of the curve. For sake of brevity, we will omit those details for now. Our system of equations might have a greater or smaller number of form parameters, but they will be incorporated in the same way.
Hence, an unconstrained optimization problem for $F$ is established whose unknowns are the coordinates of the remaining free vertices and Lagrange multipliers. Presuming an unbounded feasible domain, the first order necessary condition for a local extremum is

$$F^{\dagger} = \text{Minimum} \implies \nabla F^{\dagger} = 0$$

(4.23)

Thus, by differentiating the scalar functional $F$ with respect to free variables and Lagrange multipliers, we obtain a system of algebraic equations. The vector of unknowns $z$ contains the free control point coordinates of the curve, the Lagrange multipliers, and possibly slack variables. For a 2D curve using only equality constraints, the size, $N$, of this system would be $N = \text{len}(v_x) + \text{len}(v_y) + \text{len}(\Lambda)$ unknowns. Explicitly, $z$ consists of

$$z^T = (x_1, x_2, x_3, \ldots, x_{n-1}, y_1, y_2, y_3, \ldots, y_{n-1}, \lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_m)$$

(4.24)

Note that if 11 form parameters are included, the number of free variables has to be larger than 11. This requirement stipulates that there be at least six 2D vertices for each curve $(2(N = 6)) = 12 \geq 11$. More vertices increase both flexibility and numerical effort. In our formulation, we are free to choose the number of vertices, but shall typically use seven or eight vertices for a basic curve and add more when the number of constraints requires it. Most curves will be cubic B-splines (order $k = 4$) which maintain curvature continuity across curve segments unless this is deliberately changed.

From here, the problem is amenable to the techniques of deterministic optimization, with the caveat that the system of equations is generally nonlinear. Nonlinearity stems from arc length and also the form parameters for curvature and centroid location. An iterative Newton-Raphson type algorithm is employed to solve the system of coupled equations.

The algorithm is derived by writing a first order update for the values of the gradient of our scalar function. Note that we are not updating the functional here, but the gradient of the functional. Hence, the update step looks in matrix form like this:

$$\nabla F(z + \Delta z) = \nabla F(z) + \begin{bmatrix} H(F(z)) \end{bmatrix} \Delta z$$

(4.25)

Where $\Delta z = (\bar{z}_{k+1} - \bar{z}_k)$, and $k$ indicates the $k^{th}$ update, anticipating a sequence of this process. $H$ stands for the Hessian matrix of second order derivatives with respect to the components of the vector of unknowns.
A potential minimum of the augmented fairness functional $F$ is determined by the solution $\nabla F (\tilde{z} + \Delta \tilde{z}) = 0$. Applying this condition and solving Equation (4.25) for $\tilde{z}_{k+1}$ results in the well known Newton-type iteration for the solution vector:

$$\tilde{z}_{k+1} = \tilde{z}_k - \left[ H (F (\tilde{z}_k)) \right]^{-1} \nabla F (\tilde{z}_k)$$ (4.26)

Convergence is quadratic for Newton-type methods in the neighborhood of a stationary point (extreme value of our gradient). Note that the advantages of Newton’s method over other methods, such as gradient and steepest decent methods is given by [58] as

- Quadratic convergence is exhibited by the method near $z^*$, the stationary point.
- Affine invariance. This gives freedom to choose coordinate systems.
- Scaling with dimensional problem size. For a second order method, performance on large $n$ optimization spaces $R^n$ is similar to that in small $n$ spaces. In contrast, for a first order method, like a sub-gradient method, performance is highly problem scale dependent. Here Boyd suggests that Newton’s method does not suffer performance degradation when moving from low dimension problem spaces to high dimension problem spaces, in contrast with gradient driven methods.
- Performance is independent of other algorithm parameters. In contrast, the choice of pseudo-norms for steepest decent plays a critical role in performance.

Conversely, as noted by Boyd in [58], the chief disadvantages of the method are the cost of computing the Hessian matrix, and the cost of inverting it to solve a set of linear equations (or solving them through, e.g. Gauss Seidel iteration). Boyd further notes that the traditional problems of Newton’s method can be addressed by moving to the family of quasi-Newton methods. These use various tactics to reduce the cost of the full Newton’s method while retaining the swift convergence in the vicinity of a stationary point. We do not attempt such improvements at this point as the idea here is to facilitate a geometric design algorithmic experiment, not a focus on efficient optimization itself, as mentioned in Section 1.

As might be inferred from the equations and exposition above, there is a substantial programming effort required simply to code all necessary derivatives for the algorithm, in addition to coding the functions themselves. This paper introduces an implementation which side steps the work of programming derivatives entirely. Instead of writing and debugging such functions for the gradient and Hessian of $F$, the technique of
automatic differentiation (AD) is employed. A suitable, simple AD class has been implemented such that the free variables of the curve design problem are represented as AD types. The gradient $\nabla F$ and Hessian $H$ needed in [52] are computed alongside the value of the Lagrangian functional $F$. 
Chapter 5

Automatic Differentiation Scheme

Automatic differentiation (AD) is a numerically exact way to evaluate derivatives in a computer program. It is neither symbolic differentiation, nor does it use the finite differences common in PDE modeling. It combines the flexibility of those approaches with much of the computational efficiency of hand coded derivatives. The key idea is to implement the basic derivative rules and formulas from calculus in a programming environment. This will exploit the fact that the computational components of scientific codes are often composed of sequences of elementary arithmetic operations and functions. By applying the rules of calculus to these elementary computations, the values of arbitrarily complicated functions and their derivatives can be computed automatically. This is achieved via a Python class which overloads the basic mathematical functions to return not only the function, but the desired derivatives of that simple function as well. Then more complicated functions composed from these building blocks will necessarily return their derivatives in like manner.

As mentioned in the introduction, several Python packages are available for automatic differentiation. However, implementing our own automatic differentiation allowed us to control the computational effort and enabled us to optimize the interoperability with our curve design utilities.

AD methods are subdivided into forward and reverse mode types, which specify the direction in which the computation proceeds along the computational graph. For details see [86]. The AD scheme employed here is of the forward type. Although forward mode is not the most efficient for optimization problems [87], the scheme was primarily chosen for ease of implementation. For our purposes it proved “fast enough” and the elegance of implementation is more important than optimizing computational performance at this stage. Once AD is implemented for scalar derivatives, the crucial insight for multivariate functions is that all derivative rules generalize with a gradient in place of a derivative [94]. Thus for multivariate functions
the correct arrays of derivatives can be returned. This makes AD useful in an optimization context and in fact, the NumPy package for Python provides exactly the required capabilities. Thanks to dynamic typing, a NumPy array can hold objects of the AD class, and will compute any NumPy array operation, such as a dot product, with arrays of these AD objects. In the process, the overloaded types respond appropriately to each multiplication or any other required operation we have included in our AD class. In the end, these computations result in an object with a value equal to the result of the operation, a vector gradient of first partials of the function with respect to the underlying variables of the space and a Hessian square matrix which is composed of the required second partials as well.

So the design variables will be members of the AD class. Further, the AD class will contain both data structures to facilitate storage of the results of the aforementioned computations, and the overloaded computational functions themselves. An example will facilitate the understanding of both aspects as follows: Given a design space where \( N = 3 \), the vector \( \mathbf{x} \) contains three AD design variable objects:

\[
\mathbf{x} = [x_1, x_2, x_3]
\]  

(5.1a)

Each scalar in the vector space shall be it’s own separate AD object. Following [94] we require that each AD object stores its own gradient vector. This gradient is the vector of 1st partial derivatives of itself with respect to all variables in the space. So we have three AD objects, \( x_1 \), \( x_2 \), and \( x_3 \) which store their own gradients. For the gradient of \( x_1 \) we have:

\[
x_1.\text{grad} = \left[ \frac{\partial x_1}{\partial x_1}, \frac{\partial x_1}{\partial x_2}, \frac{\partial x_1}{\partial x_3} \right] = [1, 0, 0]
\]

(5.1b)

Please note that this is not the gradient of the vector \( \mathbf{x} \).

The equivalent expressions for the 2nd and 3rd design variable are:

\[
x_2.\text{grad} = \left[ \frac{\partial x_2}{\partial x_1}, \frac{\partial x_2}{\partial x_2}, \frac{\partial x_2}{\partial x_3} \right] = [0, 1, 0]
\]

(5.1c)

\[
x_3.\text{grad} = \left[ \frac{\partial x_3}{\partial x_1}, \frac{\partial x_3}{\partial x_2}, \frac{\partial x_3}{\partial x_3} \right] = [0, 0, 1]
\]

(5.1d)
The objects also store a copy of the matrix of second derivatives, the Hessian:

\[
x.1\text{.hess} = x.2\text{.hess} = x.3\text{.hess} = \\
\begin{bmatrix}
\frac{\partial^2 x_1}{\partial x_1^2} & \frac{\partial^2 x_1}{\partial x_1 \partial x_2} & \frac{\partial^2 x_1}{\partial x_1 \partial x_3} \\
\frac{\partial^2 x_2}{\partial x_2 \partial x_1} & \frac{\partial^2 x_2}{\partial x_2^2} & \frac{\partial^2 x_2}{\partial x_2 \partial x_3} \\
\frac{\partial^2 x_3}{\partial x_3 \partial x_1} & \frac{\partial^2 x_3}{\partial x_3 \partial x_2} & \frac{\partial^2 x_3}{\partial x_3^2}
\end{bmatrix}
\]

\[(5.1e)\]

Where of course equality in this case means only equality of value. To summarize the AD scheme, the AD object \(x_1\) would actually be an object storing the following three pieces of data:

1. \(x_1\) itself
2. the gradient of \(x_1\), that is (5.1b)
3. a square null matrix of size equal to the design space, (5.1e)

Every primitive design variable in the design vector \(\mathbf{x}\) is defined in this way, with a given value, which will change during an optimization, a “gradient,” which is fixed with respect to the variables of the design space, \(n\), and Hessian, which is simply a null square matrix of size \(n \times n\).

To avoid confusion here, Equations (5.1b), is a restatement of the fact that the derivative of \(x_1\) with respect to \(x_1\) is 1, and with respect to \(x_2\) and \(x_3\), it is zero. A similar pattern is followed for Equations (5.1c) and (5.1d) as well. For purposes of automatic differentiation, these simple facts must be stored in an array. This concludes the data structure portion of the AD definition.

As mentioned before, our AD class must also overload the necessary mathematical functions. The basic design variables can then be combined with these overloaded functions to assemble any function that is needed. Naturally any composite function will return a new AD object with the correct gradient and Hessian with respect to the primitive design variables of the space. The code example in Listing 5.1 demonstrates how this is programmed by using the basic rules of calculus to compute the value, gradient (Jacobian), and Hessian for the functions in our AD class (which here is named AD). Only a few of the elementary functions are shown here.
class AD(object):
    # Construct the Class Instance:
    def __init__(self, value=None, der=None, hess=None):
        self.value = value
        self.grad = der
        self.hess = hess
    # Overload addition:
    def __add__(self, other):
        return AD(self.value + other.value,
                  self.grad + other.grad,
                  self.hess + other.hess)
    # Overload multiplication:
    def __mul__(self, other):
        return AD(self.value * other.value,
                  self.grad * other.value + self.value * other.grad,
                  self.hess * other.value
                  + np.transpose(self.grad) * other.grad
                  + np.transpose(other.grad) * self.grad
                  + other.hess * self.value)
    # Overload sine:
    def sin(self):
        return AD(np.sin(self.value),
                  np.cos(self.value) * self.grad,
                  -np.sin(self.value) * np.transpose(self.grad) * self.grad
                  + np.cos(self.value) * self.hess)

Application of the class is illustrated with an example. We intend to compute the sine of the product of two variables \( \sin(x_1 x_2) \). The first step is the instantiation of the free variables as objects of type AD. The code in Listing 5.2 performs the following operations:

- the dimensions of the space are set to 2.
- \( x_1 \) is set to 0.5 and \( x_2 \) is set to 1.0
- Identity matrix \( \text{Imatrix} \) is created with size equal to rank of the design space \( N = 2 \)
- \( x_1 \) and \( x_2 \) are converted into AD objects with gradients equal to the first and second row of the identity matrix, respectively, and a null Hessian of dimension \( N \times N \).
Source Code 5.2: Instantiation of AD numbers in 2D design space

```python
N = 2  # set dimension of the "design space"
x1 = 0.5  # set the value of the first variable
x2 = 1.0  # set the value of the second variable

Imatrix = np.identity(N)  # the Numpy identity matrix encodes the gradient.

# convert the variables into AD objects
x1 = AD(x1, np.matrix(Imatrix[0]), np.matrix(np.zeros((N, N), float)))
x2 = AD(x2, np.matrix(Imatrix[1]), np.matrix(np.zeros((N, N), float)))
```

Equations (5.2a–5.2c) show what the AD routine actually computes when tasked with evaluating the expression \( \sin(x_1x_2) \).

\[
\text{value} = \sin(x_1x_2) \tag{5.2a}
\]

\[
\text{grad} = \begin{bmatrix}
\cos(x_1x_2) \frac{d(x_1x_2)}{dx_1} \\
\cos(x_1x_2) \frac{d(x_1x_2)}{dx_2}
\end{bmatrix} \tag{5.2b}
\]

\[
\text{hess} = \begin{bmatrix}
0 - x_2^2 \sin(x_1x_2) & \cos(x_1x_2) - x_1x_2 \sin(x_1x_2) \\
\cos(x_1x_2) - x_1x_2 \sin(x_1x_2) & 0 - x_1^2 \sin(x_1x_2)
\end{bmatrix} \tag{5.2c}
\]

Our class in Listing 5.1 basically implements linear algebra with some elementary rules of differentiation. Then when the program computes \( \sin(x_1x_2) \), all of the derivative structure is computed as well. These results are then stored in a new AD variable and returned to the caller. We can quickly verify the accuracy of these calculations. After instantiating \( x_1 \) and \( x_2 \) as in Listing 5.2, executing the line

```python
s1 = (x1*x2).sin()
```

will return the results:

- \( s1.\text{value} = 0.479425538604 \)
- \( s1.\text{grad} = [0.8775825618903726, 0.43879128094518638] \)
- \( s1.\text{hess} = [[-0.47942553860420301, 0.63786979258827126], [0.63786979258827126, -0.11985638465105075]] \)

Results are accurate to machine precision. The method extends to as many variables as we would like to add, and, as coded here, to any functions whose first and second derivatives we care to enter into the automatic differentiation class.
B-spline Evaluation Example

It may be instructive to show how this scheme would work in the case of B-spline curve evaluation. From Section 4, we have the B-spline evaluation rule,

\[ q(t) = \sum_{i=0}^{n} V_i N_{i,k}(t) \]  

Equation (5.3) is now to be computed with AD control vertices. \( V \) is a vector of \( n + 1 \) two dimensional (2D) AD control vertices. For this example, assume \( n = 6 \), that is, there are seven 2D AD vertices.

For our purposes of demonstration here, assume the B-spline basis functions at a given parameter location have already been computed. We must now instantiate the free variables (control vertices) as objects of type AD. If this is done correctly, evaluation will become as simple to program as the dot product on vectors of real numbers.

Accordingly, the details of instantiation of the control vertex components as AD objects are of primary importance. We will need a separate AD object for each degree of freedom within each control vertex. Some care is needed in the construction of the gradient: In accord with vector calculus, the gradient of each AD object with respect to all the other degrees of freedom of the design space (all the other control vertices’ components) will be 0 everywhere except where the partial derivative is taken with respect to that particular degree of freedom itself. Naturally, the Hessian of an AD primitive is the \( N \times N \) null matrix, where \( N \) is the number of dimensions of the design space. The code in Listing 5.3 performs the following operations:

- Since there are seven 2D vertices, the rank of the design space is \( N = 14 \). Our intended curve will exist in 2D space, but in terms of optimization, each of the independent variables of a control point represents a degree of freedom in which the design can be changed. Further, the gradient and Hessian of our design vector stipulate that each component of the design vector is independent of the others. This is what mathematically defines each variable as belonging to a different dimension of the design space.

- Assume that \( x \) and \( y \) values have been initialized elsewhere. Now we are concerned with transforming them into AD variables suitable for use in our optimization problem.

- Identity matrix \( \text{Imatrix} \) is created with size equal to rank of the design space \( N = 14 \). Of course in a real optimization problem there will be form parameters with associated Lagrangian multipliers. These will increase the rank further.

- \( v_x \) and \( v_y \) are NumPy arrays of AD objects.
• Each degree of freedom is an AD variable with an independent gradient. That is, it will have a gradient that is 0 with respect to all other degrees of freedom, and 1 with respect to itself.

• The design space is arranged such that the first “x” variable has a gradient with 1 in the first position, and 0 elsewhere.

• The next next variable is the second “x”. This has a gradient with a 1 in the second position, and 0 elsewhere.

• The “x” terms continue until they terminate with the final “x” variable containing a gradient with a 1 in only the 7th position.

• The “y” terms come next in the sequence. The first “y” term will have a gradient with the 1 in the 8th position and 0 elsewhere.

• This continues until the final “y” term which has a 1 only in the final position of the gradient vector and 0’s elsewhere.
Source Code 5.3: Instantiation of AD numbers part 2: Setup for use with Python’s Numpy Package

N = 14  

# set dimension of the “design space”  
# of 7 control vertices in 2D.

# Find a set of 7 starting values in x:  
ini_x_values = [x1, x2, x3, x4, x5, x6, x7]

# Find a set of 7 starting values in y:  
ini_y_values = [y1, y2, y3, y4, y5, y6, y7]

Imatrix = np.identity(N)  

# the NumPy identity matrix encodes the gradient.

xlist = []  
# initialize storage for the AD variables in the X direction

ylist = []  
# initialize storage for the AD variables in the y direction

# Give each AD variable a valid gradient, a vector  
# linearly independent from the rest  
# and a valid hessian matrix, of null values.

for i, val in enumerate(ini_x_values):
    xlist.append(AD(value = val,  
                    grad = np.matrix(Imatrix[i]),  
                    hess = np.matrix(np.zeros((N,N), float))))

# Importantly, the gradient of the y variables continues where  
#
#
pos = i  
for i, val in enumerate(ini_y_values):
    j = i + pos
    ylist.append(AD(value = val,  
                    grad = np.matrix(Imatrix[j]),  
                    hess = np.matrix(np.zeros((N,N), float))))

# we may ustilize NumPy’s dot product if we store the  
# lists above as NumPy arrays:

Vx = np.asarray(xlist)
Vy = np.asarray(ylist)

The pseudo-code shown in Listing 5.3 demonstrates how AD objects representing B-spline control vertices are arranged into NumPy arrays for use in B-spline computations. Computing the dot product result in an AD object with the value, gradient, and Hessian of the B-spline curve with respect to its control vertices, exactly.

As noted above, the basis functions may be pre-computed and stored for a set of parameter values $t$. Assume this has been done using the algorithms given in [24] and the vector $N_k(t)$ is stored as a 1D
NumPy array. Let’s assume \( t = 0.5 \) for this example. For a cubic B-spline basis for \( n + 1 = 7 \) vertices, NumPy produces the following real valued vector.

\[
N_k(0.5) = \begin{bmatrix}
0.
0.
0.16666667
0.66666667
0.16666667
0.
0.
\end{bmatrix}
\]

Each component represents the value of a basis function at \( t = 0.5 \).

Equations (5.4) show what the AD routine actually computes when it evaluates the expression \( \sum_{i=0}^{n} V_i N_{i,k}(t) \). In the program, we simply ask NumPy for the dot product of \( v_x \) with \( N_k \) to get the \( x \)
component of the curve:

\[ q_x = \sum_{i=0}^{n} x_i N_{i,k}(t) = v_T^T N_k(t) \]  \hspace{1cm} (5.4a)

\[ q_x \text{.value} = \text{equivalent to standard B-spline evaluation} \]  \hspace{1cm} (5.4b)

\[ q_x \text{.grad} = \begin{bmatrix}
0. \\
0. \\
0.16666667 \\
0.66666667 \\
0.16666667 \\
0. \\
0. \\
0. \\
0. \\
0. \\
0. \\
0. \\
0.
\end{bmatrix} \]  \hspace{1cm} (5.4c)

\[ q_x \text{.hess} = \text{Null square matrix} \]  \hspace{1cm} (5.4d)

For the \( y \)-component we take the dot product of \( \mathbf{V}_y \) with \( N_k \).

B-splines are linear with respect to the vertex coordinates. Therefore the partial derivatives with respect to the vertex coordinates (free variables in the augmented Lagrangian function) return the B-spline function values as is evident in Equation (5.4c) above. The first seven components of the gradient are for the derivatives with respect to the \( x \)-coordinates of the vertices, and the remaining seven components are for the derivatives with respect to the \( y \)-coordinates which of course do not play a role in \( q_x \). Since the first order derivatives are constants, the Hessian is a \([14 \times 14]\) null matrix.

In the actual form parameter design, gradients and Hessians will also include the derivatives with respect to the Lagrangian multipliers and the slack variables. It should also be emphasized that the parametric derivatives of the basis functions which are needed for the computation of tangents, curvature, area, etc. are computed with de Boor’s B-spline differentiation algorithm [24]. With our AD class in hand, we will now apply it to the design of B-spline curves.
Chapter 6

Automatic Differentiation of B-splines

This section explains how the automatic differentiation (AD) of Section 5 is tied together with Lagrangian B-spline optimization outlined in Section 4 into the solution of form parameter driven curve design problems. We must account for curve initialization, design variable instantiation for AD, computation of the Lagrangian functional and its derivatives, and the Newton-type iteration with convergence checking which solves the necessary condition for a minimum of the Lagrangian functional. The end result is a fair B-spline curve that satisfies the desired form parameters. The five steps are outlined below.

The augmented fairness functional (4.22) must be differentiated twice with respect to all design variables and Lagrange multipliers. Fortunately, most of the tedious programming work is actually taken care of by the formulation of the AD class (Section 5). Again, more efficient and more general implementations are possible [87]. However, with only first and second order derivatives needed and the comparatively small number $N$ of unknowns the tool developed here is adequate for our curve design purposes.

1. Initial curve design:

Iterative solution of the generally nonlinear necessary condition for a minimum of the Lagrangian functional requires an initial solution. This suitable initial curve need not satisfy all constraints but should represent a “good guess” at fulfilling the form parameters. The initial curve, however it is generated, is built with standard $n + 1$ B-spline control vertices $V_i = (x_i, y_i)^T$ with $i = 0, 1, 2, \ldots, n$.

We use the method proposed by Harries [52] to create an initial set of control vertices from form parameters within the curve solver. \footnote{Note this is distinct from the greater scope of this thesis - where we will be using other (logical, interval) methods to generate the target form parameters themselves.} Additionally, we solve a Lagrangian functional for a curve with some of the constraints, and then add additional constraints to produce a more complex form. This
second method is particularly useful after adding new constraints to the optimization problem.

2. Instantiation of design variables \((x_i, y_i)\) and Lagrange multipliers \(\lambda_i\) in AD:

Instantiation proceeds much as in Listing 5.3, with some simple changes to accommodate the Lagrange multipliers and structure of the problem. For our purposes, the solution vector \(\mathbf{z}\) (4.24) will contain first all the \(x_i\)-coordinates of vertices, then all the \(y_i\)-coordinates of vertices, and finally all of the \(m\) Lagrange multipliers \(\lambda_i\). Note that the first and last vertex \(V_0\) and \(V_n\) are known from the form parameters and do therefore not appear in the solution vector \(\mathbf{z}\). The dimension of the solution vector is

\[
N = 2(n - 1) + m
\]

(6.1)

Instantiation starts with the creation of \(N\) objects of the \(\text{AD}\) class. The objects are collected into a NumPy array \(\mathbf{z}_0\). Actual values are taken from the initial control point vertices and Lagrange multipliers of constraints are set equal to one. The gradient of each solution variable \(z_j\) corresponds to the respective row vector \(j\) in an identity matrix \(I\) of size \([N \times N]\). The Hessians are matrices of size \([N \times N]\) which are set to zero.


With variables instantiated, the stage is set to compute the automatic differentiated fairness functional of choice. All components making up the functional, from solution variable to functions used, are objects of type \(\text{AD}\). Consequently, the functional value \(F(\mathbf{z}_k)\) is accompanied by its gradient \(\nabla F(\mathbf{z}_k)\) and Hessian matrix \(H(F(\mathbf{z}_k))\). No additional steps are necessary to compute gradient or Hessian of \(F\).

4. Newton update:

Based on the Newton iteration (4.26) an updated solution \(z_1\) is derived. This involves inversion of the Hessian matrix which is accomplished with standard linear algebra algorithms for matrix inversion. Now steps 3 and 4 are repeated, increasing the iteration counter \(k = 1, 2, \ldots\) until the \(\ell_2\)-norm of the gradient of the Lagrangian functional \(c = \nabla F\) becomes smaller than a pre-defined, small tolerance \(\varepsilon\).

\[
\ell_2(c) \equiv |c| \equiv \sqrt{\sum_{j=1}^{N} |c_j|^2} < \varepsilon
\]

(6.2)
The final solution vector $z$ is assumed to minimize the Lagrangian functional $F$. Its vertex coordinates define a B-spline curve which is fair and satisfies the form parameters defined by the user.

\footnote{We assume we have a minimum. All we know is that we have a critical point. We could check second order sufficiency criteria but in practice for ship hull design, we only care about whether or not the geometry matches with our engineering purpose. That is what we will check, e.g. does it meet the constraints, are the curves sensible, etc.}
Chapter 7

B-spline Curve Design Examples

Some examples shall demonstrate the utility of an AD Lagrangian curve optimization algorithm with regards to curve design for ship hull forms. For complicated curves, best results have been obtained by starting with a reduced number of constraints, initialized via the method of Harris in [52], and progressively adding constraints to the initial curve solution until the desired result is reached. In this way, more complicated curve forms could be built without experiencing solver divergence. Please note that in all examples, fairness functions and approximate arc length function are scaled by a factor of 0.5. The fairness pseudo-norms $E_1$, $E_2$, $E_3$, and the approximate arc length function described in section 4 are used throughout the following examples. No attempt was made to search for optimal scaling values of these functions, as the purpose of the AD sections of this thesis is to describe how AD makes the optimization framework simple to construct, and not to find any particular optimal result. Nevertheless, the examples below are important as demonstrators of the utility and success of the methods outlined in the earlier sections of this paper. As mentioned earlier, knot vectors will be fixed so that the curve is endpoint interpolating, with uniform knot intervals on the interior of the knot vector. In this way the effect of control point manipulation will never be pitted against knot vector transformation within the optimization loop. This increase in efficiency comes at the expense of forcing the algorithm to enforce continuity conditions in explicit geometric form as tangents, curvature and position. Fortunately this is what we typically want as naval architects - as ship hull designers. Continuity is important, especially at the boundaries between curves, but it is often better described in geometric constraint terms within the curves. Then computational procedures outside of the optimization loop can be used to stitch curves together, or split them apart, if need be.

As a first example we present the design of a sectional area curve (SAC) of a ship. The sectional area curve describes the volume distribution over the length of the ship. The SAC curves presented below are
single B-spline curves. This “single curve” method is a departure from common practice where a SAC curve is created by subdividing the curve into a number of sections and solving for them separately. In the past, this has been done in order to allow local constraints, such as tangent and curvature, to be set at intermediate locations along the length of the SAC curve. However, subdivision of the SAC curve adds to the complexity of enforcing global hull properties. Our solver allows for specifying tangent and curvature constraints at any location along the length of the curve, eliminating the need to split SAC into multiple segments in this instance.

### SAC Design and Optimization

For the design of a sectional area curve (SAC), sometimes a smoothly curving ($C_2$ continuous) form is desired, as with say, a classic sailing yacht, or a simple vessel such as a canoe body, and other times a curve with a flat midsection (parallel mid-body) SAC produces the best compromise between production efficiency and fairness of the ship hull itself. Note that the SAC defines the area of a transverse slice of the hull form at any section. Accordingly, the smoothness of the SAC translates into smoothness of volume variation in the actual hull.

In Figure 7.1, a simple SAC curve is designed by form parameter optimization. Constraints are set for area, $A$, moment, $X_c$, and starting and ending tangent angles $\alpha_B$ and $\alpha_E$. Note also that $q_x$ and $q_y$ denote curve interpolation constraints in this figure. Such interpolation constraints allow for specific hull features, such as flat of side (this term refers to an portion of the vessel with straight sides parallel to the its centerline),
to be implemented between particular points chosen along the hull. Interpolation form parameters can also have an effect on the details of the area distribution. Form parameters for this plot are given in Table 7.1.

In Figure 7.2a, the SAC curve in Figure 7.1 was modified to incorporate a flat mid section with a certain longitudinal center of area, $X_c$, as well as to have zero curvature at the end points. Note that curvature spines are shown in this plot scaled up by a factor of 3.5. Form parameters for this plot are given in Table 7.2.

In Figure 7.2b, the SAC curve in Figure 7.2a was modified by shifting the center of area while holding all other form parameters constant. This had a significant impact on the curvature. As prescribed, it is zero at the endpoints, but in transition areas, it increases to amounts that are greater than those seen in Figure 7.2a. Here it is again scaled by a factor of 3.5 for visibility. Form parameters for this plot are given in Table 7.3.

All SAC curves were designed as a single entity so that there was no need to keep track of the area and
### Form parameters of initial SAC curve

<table>
<thead>
<tr>
<th>Form Parameter</th>
<th>Symbol</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>Δ (2) − (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$ Area of SAC curve, (Volume of Ship)</td>
<td>$A$</td>
<td>288.0</td>
<td>288.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_2$ LCB (Longitudinal Center of Buoyancy),</td>
<td>$x_c$</td>
<td>18.0</td>
<td>18.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_3$ Tangent of Entrance</td>
<td>$\alpha_B$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_4$ Tangent of Run</td>
<td>$\alpha_E$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_5$ Curve y Interpolation Constraint</td>
<td>$q_{y_1}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_6$ Curve y Interpolation Constraint</td>
<td>$q_{y_2}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 7.1: Form parameters of initial SAC curve, Figure (7.1)

### Form parameters of SAC curve with flat portion

<table>
<thead>
<tr>
<th>Form Parameter</th>
<th>Symbol</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>Δ (2) − (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$ Area of SAC curve, (Volume of Ship)</td>
<td>$A$</td>
<td>288.0</td>
<td>288.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_2$ LCB (Longitudinal Center of Buoyancy),</td>
<td>$x_c$</td>
<td>18.0</td>
<td>18.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_3$ Tangent of Entrance</td>
<td>$\alpha_B$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_4$ Tangent of Run</td>
<td>$\alpha_E$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_5$ Curve x Interpolation Constraint</td>
<td>$q_{x_1}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_6$ Curve x Interpolation Constraint</td>
<td>$q_{x_2}$</td>
<td>24.0</td>
<td>24.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_7$ Curve y Interpolation Constraint</td>
<td>$q_{y_1}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_8$ Curve y Interpolation Constraint</td>
<td>$q_{y_2}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_9$ Curvature of Entrance,</td>
<td>$c_B$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{10}$ Curvature of Run,</td>
<td>$c_E$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{11}$ Tangent at $q_{y_1}$</td>
<td>$\alpha_1$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{12}$ Tangent at $q_{y_2}$</td>
<td>$\alpha_2$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{13}$ Curvature at $q_{y_1}$,</td>
<td>$c_1$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{14}$ Curvature at $q_{y_2}$,</td>
<td>$c_2$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 7.2: Form parameters of SAC curve with flat portion
Form parameters SAC curve with flat portion and centroid of area shifted

<table>
<thead>
<tr>
<th>Form Parameter</th>
<th>Symbol</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>∆ (2) − (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$ Area of SAC curve, (Volume of Ship)</td>
<td>$A$</td>
<td>288.0</td>
<td>288.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_2$ LCB (Longitudinal Center of Buoyancy), $x_c$</td>
<td>16.0</td>
<td>16.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>$h_3$ Tangent of Entrance</td>
<td>$\alpha_B$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_4$ Tangent of Run</td>
<td>$\alpha_E$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_5$ Curve x Interpolation Constraint</td>
<td>$q_{x_1}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_6$ Curve x Interpolation Constraint</td>
<td>$q_{x_2}$</td>
<td>24.0</td>
<td>24.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_7$ Curve y Interpolation Constraint</td>
<td>$q_{y_1}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_8$ Curve y Interpolation Constraint</td>
<td>$q_{y_2}$</td>
<td>12.0</td>
<td>12.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_9$ Curvature of Entrance,</td>
<td>$c_B$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{10}$ Curvature of Run,</td>
<td>$c_E$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{11}$ Tangent at $q_{y_1}$</td>
<td>$\alpha_1$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{12}$ Tangent at $q_{y_2}$</td>
<td>$\alpha_2$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{13}$ Curvature at $q_{y_1}$,</td>
<td>$c_1$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_{14}$ Curvature at $q_{y_2}$,</td>
<td>$c_2$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 7.3: Form parameters of SAC curve with flat portion. Centroid of area shifted.

moments of various sections to ensure that the overall curve had the total area and moment desired. Instead, the curve is specified to have a flat section, area, and moment all at the same time, and the solver is able to find the solution.

**Design of a Stern Profile Curve**

To show some of the complicated geometries that can be developed with the optimization framework given here, in Figure 7.3 the design of the run profile of a container ship is given. In particular, relative constraints are used. These are employed to enforce a relative offset between two control points. Other geometric relations might be imagined in the future.

The shaded area is specified by the area constraint. The stern shape shown in Figure 7.3, is modeled as a single curve, including the flat section where the propeller shaft pierces the hull and the hard cusp in the upper section of the curve. Again, the technique of starting with a simple curve and adding more constraints to previously solved curves is performed. The initial curve is a straight line through the endpoints. The solver required two passes to attain the curve shown here.

The first pass incorporated the cusp at $K_0$ and starting and ending tangency conditions. In Table 7.4, these are the first three form parameters listed. The second pass added the area constraint, tangency angle on
Figure 7.3: Run Profile (This is a single B-spline curve). Note the relative constraints, $r_1$, $r_2$, and $r_3$, as well as the particular cusps, $k_1$ and $k_2$, which enforce user chosen geometry while allowing the local assembly to remain free to move relative to the rest of the control points.

The interior of the curve, and all additional relative constraints and cusps. These are form parameters four through twelve in Table 7.4. In general more passes are sometimes needed when the geometry requirements are more numerous. This can require a bit of trial and error which might be eliminated with a higher level optimization framework that will be the topic of future research. Such an optimization framework would again use optimization to determine the best combination of geometry constraints to develop a given shape, or to find optimum shapes for a given design purpose. That is, a higher level optimization framework is one that optimizes the inputs (in this case form parameters) to a lower level framework.

The propeller hub could be designed in numerous ways. In this case, constraint $h_{10}$ fixes the lateral distance from the preceding control vertex, and the second hub cusp is specified to have a vertical offset below the initial cusp. Additional constraints $h_8$ and $h_9$ are imposed to show how the whole system of hub
<table>
<thead>
<tr>
<th>Form Parameter</th>
<th>Symbol</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>( \Delta ) (2) − (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_1 ) tangent angle at start, ( \alpha_b )</td>
<td>-90.0</td>
<td>-90.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>( h_2 ) tangent angle at end, ( \alpha_e )</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>( h_3 ) first knuckle, ( k_{0} )</td>
<td>(10.0, 0.)</td>
<td>(10.0, 0.)</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>( h_4 ) Area shown in green, ( A )</td>
<td>120.0</td>
<td>120.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>( h_5 ) tangent angle after first knuckle, ( \alpha_1 )</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>( h_6 ) second knuckle, ( k_1 )</td>
<td>free</td>
<td>free</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>( h_7 ) third knuckle, ( k_2 )</td>
<td>free</td>
<td>free</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

Relative Form Parameters Below

| \( h_8 \) \( x_6 − x_5 \) | \( r_1 \) | 0.0 | 0.0 | 0.0 |
| \( h_9 \) \( y_6 − y_5 \) | \( r_1 \) | 3.0 | 3.0 | 0.0 |
| \( h_{10} \) \( x_7 − x_6 \) | \( r_2 \) | 4.0 | 4.0 | 0.0 |
| \( h_{11} \) \( x_{10} − x_7 \) | \( r_3 \) | 0.0 | 0.0 | 0.0 |
| \( h_{12} \) \( y_{10} − y_7 \) | \( r_3 \) | 2.0 | 2.0 | 0.0 |

Table 7.4: Relative form parameters allow the solver a prescribed amount of freedom to position the stern bulb.

and surrounding vertex structure can be fixed relative to itself, and yet this assembly remains free to move globally to minimize the augmented fairness functional for the entire curve, as much as possible given this large fixed assembly.

A designer could use the relative constraints demonstrated here in many ways. For instance, a relative constraint could have been imposed that set the lateral distance from the forward end of the curve to the hub instead. Alternatively, one could specify points to be collinear with each other, or some other arrangement. Higher level geometric assemblies (that is, assemblies of primitives that have their own coherence due to the lower level relative constraints) could be specified from these more primitive relative constraints. The automatic differentiation makes this as simple as assigning another Lagrangian parameter (of automatic differentiation type) to the space, multiplying this new parameter with the new relative constraint function to be imposed, and adding the result to the overall Lagrangian functional. The equation space thereby increases in size by one variable, (this does have to be taken into account during design variable instantiation, as noted in the AD examples previously) and computation of the fairness functional automatically includes all the information required for the Newton update.
Finding a Fair Enclosure

The tools demonstrated in this paper could be used to design a hull section which encloses a certain cargo or specified machinery area. In the examples shown in this section, this area is represented by an arbitrary polygon. The curve, once optimized, shall not cross into the interior of this polygon.

Given some initial curve, forcing the curve to encompass this polygonal area is a three step process. In step one, the intersections of the curve and the polygonal area are found by binary search. In the event that the curve does not initially intersect the box, locations on the curve are found which correspond to the extrema of the box by projection in each coordinate direction. In step two, inequality constraints are imposed to prevent the curve from impinging upon the polygon at those points. The third and final step is to simply run the optimization algorithm and solve for the constrained curve. This simple method is enough to force the curve away from the polygon in simple circumstances.

The plot in Figure 7.4a was generated in this way. Due to the inequality nature of the constraints, the curve touches the boundary of the enclosure at points which are not predetermined. In fact, there is no guarantee that the curve will not violate the bounding box at some other point. This method therefore is by no means guaranteed success, but shows the flexibility of the curve design framework in aiding more complicated design scenarios. These constraints easily could be tweaked to maintain a gap between curve and enclosed area as well. Other generalizations are possible as well.
### Curve with Active Inequalities

<table>
<thead>
<tr>
<th>Form Parameter</th>
<th>Symbol</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>∆ (2) − (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>$\text{Area}$</td>
<td>72.0</td>
<td>96.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_2$</td>
<td>$\alpha_b$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_3$</td>
<td>$\alpha_e$</td>
<td>75.0</td>
<td>75.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_4$</td>
<td>$c_0$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_5$</td>
<td>$c_1$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Inequality Constraints**

- $h_6$ interpolation, $q_{y1} < 4.0$ inactive 0.0
- $h_7$ interpolation, $q_{x2} > 8.0$ active 0.0
- $h_8$ interpolation, $q_{y2} > 8.0$ active 0.0

Table 7.5: form parameters for curve in Figure 7.4a

### Curve Designed to Follow Polygon Boundary

<table>
<thead>
<tr>
<th>Form Parameter</th>
<th>Symbol</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>∆ (2) − (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>$\text{Area}$</td>
<td>72.0</td>
<td>72.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_2$</td>
<td>$\alpha_b$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_3$</td>
<td>$\alpha_e$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_4$</td>
<td>$k_1$</td>
<td>3 ctrl pts coincident</td>
<td>ok</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_5$</td>
<td>$k_2$</td>
<td>3 ctrl pts coincident</td>
<td>ok</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_6$</td>
<td>$c_1$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>$h_7$</td>
<td>$c_2$</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

**Inequality Constraints**

- $h_8$ interpolation, $q_{x1} > 4.0$ active 0.0
- $h_9$ interpolation, $q_{y1} < 4.0$ active 0.0
- $h_{10}$ interpolation, $q_{x2} > 8.0$ active 0.0
- $h_{11}$ interpolation, $q_{y2} < 8.0$ active 0.0

**Relative Constraints**

- $h_{12}$ x offset, $q_{x2} - q_{x1}$ 4.0 4.0 0.0
- $h_{13}$ y offset, $q_{y2} - q_{y1}$ 4.0 4.0 0.0

Table 7.6: form parameters for curve in Figure 7.4b
Here in Figure 7.4b, the end conditions of Figure 7.4a are modified and cusps are added at the locations of the inequality constraints. The curve is stretched to the right and the tangent angle at the end of the curve is changed to zero. Meanwhile, the area is held fixed. This forces the curve flush with the enclosed area. See Table 7.5 for the form parameters used to make this curve.

The results in this section have demonstrated that AD is compatible with CAD geometry generation by form parameter design. The goal of this exercise is not to demonstrate superiority of performance over some other method of constructing the framework. Instead, the goal has been to show conclusively that AD works for this application. With AD driven algorithms, the programmer is free to focus on creativity in application of the method, but more importantly, AD allows the programmer to quickly build his or her own implementation of form parameter design, with the primary motive being to extend it’s capabilities in unforeseen directions. Some ideas for those extensions are presented in the next section. However, instead of developing more constraints, we instead think about other ways to extend our program to automate the process of reasoning about constraint systems.

Summary

With our AD class and B-spline solver in hand, we have a flexible framework for implementing constraints at low cost in programming time.

We must next review the methodologies that will aid in us in realizing the goal of making form parameter design more robust. This new work will be divided into sections. Section 8, next, will deal with logical operations to rule out infeasible form parameters. The software will draw from both of the fields of interval arithmetic / interval analysis, and declarative programming. An interval analysis system is described first in that section, and then the constraint logic programming system will be described.

Once those descriptions are complete we will introduce the truncated hierarchical B-splines for robust optimization of individual curves, and in the process defining a surface hierarchical description as well. Only then will we move on to develop the full system, including integration with the AD curve design framework, and the addition of an interval constraint logic programming layer which will address the form parameter feasibility problem.
Chapter 8

Picking Feasible Form Parameters

Standard form parameter design begins when a selection of form parameters is entered into the form parameter system. Form parameter design systems typically receive the initial design “vector” as input, though this in turn may be part of a larger design space search in more sophisticated systems such as described by Abt and Harries [3]. For instance, form parameter design might be the geometry generation portion of a search which includes some physics simulation, or other processes. Many design optimization tools support such automation and provide means of generating, or at least modifying, many hull designs without user input. Despite many and varied types of constraint management and constraint integration systems in use with hull design software, there has been no complete work to provide the following functionality:

- Automatic generation of self-consistent form parameter design constraint sets without need to keep changes small.

- Accurate and timely suggestions for filling in design parameters not supplied by the software user, whenever possible.

- Software capable of producing self-consistent equality constraints for an entire bare hull plus extensions to more complex geometry such as a bulbous bow.

The above functionality have been desired in form parameter design systems for some time. In the PhD dissertation of Kim, [63], the problem and desired solution is stated as follows: (What is needed is) “Inference of feasible form parameters. Not all form parameters are usually known at the beginning design stage. Therefore, the naval architect either starts with a few important parameters, e.g. principle dimensions and form coefficients, and gradually builds up data by interactive modification or, alternatively, evaluates an
already existing design and employs the form parameters thus found. From this point of view, an efficient method for inferring feasible form parameters is needed for the automatic parametric design method of hull forms.”

Continuing, Kim states that various authors, such as [81, 82, 160–162] “...tried to develop efficient inference systems to determine initially feasible form parameters in hull form design. Using artificial intelligence techniques, particularly, fuzzy logic, they developed a system which can directly compute characteristic basic curves and sections from principle particulars and hull coefficients. Their application fields are however only for simple bare hull forms, and hence an efficient inference method for feasible form parameters constructing the complex ship hull forms is needed to be developed. “

The papers cited by Kim addressed feasibility indirectly by using neural nets to generate weights of fuzzy inputs which generated hull form curves. Feasibility was thus dependent on machine learning to search for an appropriate model of the feasible space.

Other researchers addressed design feasibility deterministically and logically with expert systems, e.g. [136,140,141], which include constraints and sometimes made use of intervals, but were never combined with, e.g. differentiable geometric methods for splines, or other more recently advanced geometry representations, nor combined with form parameter optimization.

Finally, Harries and Abt characterized the handling of constraints in form parameter design systems in [1,3]. There, issues of feasibility were primarily addressed through

- switching to inequality constraints
- iteration logic to continue searching after infeasible designs were encountered.

The studies noted above constitute the bulk of research carried out towards the development of more robust methods for form parameter design. To date, no research in the field of automated ship hull design has combined geometric reasoning for feasible curve and surfaces combinations, naval architecture rules, and a parametric curve generation tool for hull form generation.

In the next few sections, we will show how to construct such a system, capable of enforcing consistency across equality constraints and complex hull form geometry. In the first subsection, interval arithmetic will be introduced. The point is to introduce interval contractors, which are algorithms for bounding computations over continuous sets. They come with fixed point theorems and other capabilities for proving the existence of solutions to equations or systems of equations [6,106], and furthermore, are guaranteed by construction never
to diverge. Note that design spaces are effectively closed.\footnote{In our program, the design spaces will be closed by algorithm, since interval representations of designs will always be updated by intersection of the initial interval with any representation of it modified by new data.} Interval methods may merely fail to converge or fail to make progress narrowing the domain in the worst cases.

Interval analysis was created to bound rounding error and was soon recognized to also be suitable for attacking global nonlinear optimization problems. In practice for our system, we will only require something which, in the interval literature, is called “hull consistency,” in order to perform the necessary “interval contractor” computations on continuous domains. In the constraint logic programming section 8.2, these interval methods will be utilized in a constraint programming system that will accomplish our feasibility goals\footnote{An additional note for interval experts, we will also be using generalized division, another sort of “closure” of the system. See, e.g. Hanson and Walster [6, 114] for details. Read on for our implementation.} using constraint propragation.\footnote{It should be noted that the interval literature does also contain work on constraint propagation methods [7,107,126,127,163–168]. Since we only need standard constraint propagation techniques, extended to handle intervals, we have made the choice to keep the two fields somewhat separated.}

**Interval Analysis**

An interval is a construction which uses a pair $[a, b]$ of floating-point numbers to represent an interval of real numbers $a \leq x \leq b$. Thus an interval is a kind (or in our case, a python class) of number pair. An interval is also a set $[a, b] = \{ x : a \leq x \leq b \}$. Interval arithmetic combines set operations on intervals with interval function evaluation to get algorithms for computing enclosures of sets of solutions to computational problems [102]. So intervals are represented not by mere “endpoints,” but at the same time these endpoints are rigorous bounds on the true infimum and supremum of the interval set representing the quantity being processed.

By solving equations over the intervals instead of over the reals, a design program can reason about a design space in finite but continuous partitions (sometimes called “boxes” in the interval terminology) instead of being forced to reason about the local neighborhood of one single point (or set of a finite number of points) in the design space.

The primary purpose of using interval arithmetic and interval analytic techniques in this thesis is to

---

1. In our program, the design spaces will be closed by algorithm, since interval representations of designs will always be updated by intersection of the initial interval with any representation of it modified by new data.
2. An additional note for interval experts, we will also be using generalized division, another sort of “closure” of the system. See, e.g. Hanson and Walster [6, 114] for details. Read on for our implementation.
3. It should be noted that the interval literature does also contain work on constraint propagation methods [7,107,126,127,163–168]. Since we only need standard constraint propagation techniques, extended to handle intervals, we have made the choice to keep the two fields somewhat separated.
4. In “Contractor Programming,” [169] Jaulin and Chabert draw the distinction between contractors and propagators such that the term propagator is always associated to a constraint and is destined to be called in a propagation loop, whereas a contractor is more closely associated with root finding - where a system may not necessarily even be constrained. For our purposes though, we are using contractors to enforce consistency at the most local level. As we will see, we “contract” by taking the intersection of an interval update with itself. The common result is a smaller width for the interval being contracted, and this will spread through the constraint net, enabling the contraction of other intervals, via propagation.
quickly reason about the geometric design space of our hull. This will be realized by the following choices:

1. Each design parameter, which formerly would have been a purely real valued input to a basic form parameter design system, is now more primarily viewed as a dimension of the computational design space.

2. In accordance with this shift in perspective, each design parameter will obtain an interval representation which spans the entire width of the design space in that dimension. This width may initially be chosen by the user but it will be narrowed to the feasible range automatically by the program. Alternatively, the program may supply the feasible domain without any input from the user.

3. Each parameter will continue to be used as an input constraint to initiate the form parameter design process. As a result of the changes above, however, each interval parameter must be narrowed, or contracted, to a real valued constraint for use in form parameter design itself.

From this point of view, it becomes more tractable to reason about the inter-relationships between the form parameters in a systematic manner. Particularly, given some equations relating the various form parameters, and some interval valued design space representation of the same parameters, we can employ interval arithmetic and interval analysis to reason about which portions of any given form parameter’s design interval might be infeasible. We will see in the section below how to use interval methods to prove which regions of an interval are infeasible, and thus exclude such regions from our design space without further computation.

In this paper, we will not make use of the majority of more powerful interval analysis techniques, for existence proving, or for root (extrema) finding, but refer the reader to the extensive interval literature for further information on those interval techniques beyond the scope of this paper [6, 102, 103, 106, 109]. To use these methods would take us further into interval analysis, to use interval optimization methods such as interval Newton’s method, and derivative analysis to find and prove the existence of zeros within sub-intervals of the total design space. Such methods would constitute a powerful and interesting extension of the methods shown here, but these additional methods could not standalone for our purposes. They would in fact need to make use of more primitive methods, such as those to be shown here, in order to make progress at certain points in the design space search. Thus, the interval consistency methods here must be developed either way, and since they proved sufficient for our purposes, these additional methods were not pursued further.

So what then, are the more primitive methods that must be developed? In this paper, we focus on the necessary interval operations needed to implement interval constraint propagation. Interval constraint
propagation is a set of techniques for enforcing consistency across an interconnected graph of constraints. Using such techniques is known as constraint programming. And in this process, we will use interval contractors to compute and enforce local constraints on the design space. Constraint programming is a model of computation in which values are deduced whenever possible from locally available information. That is, values are deduced from the surrounding values. One may visualize a constraint “program” as a network of devices (relations) connected by wires. Data values may flow along the wires, and computation is performed by the devices. With few exceptions, a device computes using only locally available information, and places newly derived values on other, locally attached wires. In this way computed values are propagated. Intervals provide the evaluation procedures to make such computations work on continuous real valued data, as approximated by finite machine, or typically, floating-point, arithmetic [170]. Interval constraint propagation ensures that constraints are satisfied. This is closely tied to global optimization and nonlinear systems of equations, but is also related to core subject areas of computer science. The technique involves explicitly using the interrelationships among intermediate quantities in evaluation of algebraic expressions in nonlinear systems, objective functions and constraints [109].

In order to properly describe the structures involved in this kind of system, we must first describe enough of the basics of interval arithmetic and interval analysis. Then we may cover the essentials of constraint logic programming and combine the two into the final system for enforcing feasibility for form parameter design by filtering the design space.

Interval Notation

Interval notation in this paper will follow the standards proposed by Kearfott, et al. in [157]. Following Kearfott, a box \( x \) is the (nonempty) set of points between its lower and upper bound,

\[
x = \{ x \in \mathbb{R}^n \mid a \leq x \leq b \},
\]

so that a vector \( x \in \mathbb{R}^n \) is contained in a box \( x \), i.e., \( x \in x \), iff \( a \leq x \leq b \). Similarly, a thin box \( x = [a, b] \) (i.e., a box of zero width) is usually identified with the unique point \( x \) it contains. The concept of a thin interval comes up often in the system developed here, as “nearly thin” intervals will be chosen during construction of a feasible set of form parameters. An arbitrary point in a box \( x \) is often denoted by \( x \) or \( \hat{x} \). The set of vertices of a box \( x \) (or more generally a polytope \( S \)) is denoted by \( \text{vert} x \) (vert \( S \)).
We write $\inf x := \underline{x}$ for the lower bound, $\sup x := \overline{x}$ for the upper bound, and $\dim x = n$ for the dimension of $x$.

A (real, closed, nonempty) interval is a 1-dimensional box, i.e., a pair $x = [\underline{x}, \overline{x}]$ consisting of two real numbers $\underline{x}$ and $\overline{x}$ with $\underline{x} \leq \overline{x}$. The set of all intervals is denoted by $\mathbb{IR}$. A box $x$ may be considered as an interval vector $x = (x_1, \ldots, x_n)^T$ with components $x_k = [\underline{x}_k, \overline{x}_k]$. For example, if $\underline{x} = (\frac{1}{3})$ and $\overline{x} = (\frac{2}{4})$ then $x = \left(\frac{1}{3}, \frac{2}{4}\right)$. We will not need interval matrices in this thesis, but you may consult Kearfott [157] for such additional details. They would come in quite handy if we were to extend the system with more sophisticated methods.

Interval Relations, Operations, and Expressions

From Kearfott [157] again, “A relation $x \circ y$ (with $\circ \in \{=, <, \leq, >, \geq\}$ between two boxes $x$ and $y$ is defined to be true iff $x \circ y$ for all $x \in x, y \in y$.’’ Note here I have replaced Kearfott’s $\omega$ with $\circ$. Kearfott further notes that “Operations (and elementary functions) are automatically interpreted as the natural operations on the class of objects involved; i.e., real for real (vector, matrix) arguments, interval if an argument is interval (vector, matrix).” Lastly, when doing arithmetic to finite precision, the standard interval operations are outward rounded. To put it simply, the lower bound is rounded “down” and the upper bound is rounded “up” so that the interval containment representation rigorously bounds the true interval. Note that there is an IEEE standard for the implementation of outward rounding [171].

Definition of Interval Arithmetic

Hansen and Walster, [6] define interval arithmetic in the following way: Let $+, -, \times, \text{ and } \div$ denote the operations of addition, subtraction, multiplication, and division, respectively. If $\circ$ denotes any one of these operations for arithmetic on real numbers $x$ and $y$, then the corresponding operation for arithmetic on interval numbers $x$ and $y$ is

$$x \circ y = \{x \circ y \mid x \in x, y \in y\}$$

Where $x$ and $y$ are intervals as given in by the definitions of Kearfott, et al., summarized above. Thus, the interval value of the operation $x \circ y$ contains every possible number that can be formed as $x \circ y$ for each $x \in x$ and each $y \in y$. 
Practicalities: Interval-Valued Extensions of Real Functions

By the definition, we can see that an interval valued function \( f(x) \) would be the result of taking the set of all values computed as \( x \) varies through the entirety of the set included within \( x \). This procedure would give the best possible interval enclosure of a function at maximum expense. Clearly some approximation is needed. Here, instead, we extend a given real-valued function \( f \) by applying its formula directly to interval arguments [102].

An interval valued inclusion function is a function where,

\[
\forall x \in \mathbb{IR}^m \quad f(x) \in f(x)
\]

It turns out that not only does this work, it gives us crucial properties for guaranteeing the ability of interval analysis methods as applied to proving the existence of zeros of systems of equations, optimization problems, and the like.

Interval Analysis

With the introduction and basic working definitions of intervals sketched in enough detail, we can move on to the capabilities that we will make use of to ensure feasible starting sets of form parameters. As we’ve seen above, interval operations are more involved than those on real numbers, or on pairs of real numbers. This section motivates the extra effort, and supplies details of the computations required.

Methods

From Hansen and Walster’s text, “Global Optimization Using Interval Analysis” [6] they state that, Until fairly recently, it was thought that no numerical algorithm can guarantee having found the global solution of a general nonlinear optimization problem [6]. Various authors have flatly stated that such a guarantee is impossible. Their argument was as follows: Optimization algorithms can sample the objective function and perhaps some of its derivatives only at a finite number of discrete points. Hence there is no way of knowing whether the function to be minimized dips to some unexpectedly small value between sample points. In fact the dip can be between closest possible points in a given floating point number system.

This is a very reasonable argument, and it is probably true that no algorithm using standard arithmetic
will ever provide the desired guarantee. However, interval methods do not sample the points. They compute bounds for functions over a continuum of points, including ones that are not finitely representable. [6]

With interval analysis, a computer can deterministically find and prove the optimality of solutions to nonlinear optimization problems. Intervals provide the needed means for handling the continuum in a finite manner, on a digital computer. Several powerful special purpose algorithms, such as interval linear system inversion, interval Newton’s method and box consistency, subpaving, branch and bound, and branch and prune, etc., have been developed by researchers to facilitate the use of intervals for global optimization problems. Interval techniques extend from topological and symmetry based methods, to analytic and differential algorithms, and logic tests.

Nevertheless, as alluded to before, this thesis shall ignore the majority of these interval techniques which might have been brought to bare on the problem at hand. Instead, we follow the advice given at the beginning of in [128] — to build interval constraint solvers from both low level primitive and sophisticated solving techniques. This seems to be the most reasonable and practical way to proceed. Furthermore, in practice one builds up complex systems by first developing simple proofs of concept. Accordingly, we build only the simple solver for the more primitive constraints in this thesis. Our simple solver will demonstrate the basic utility of the ideas. More flexible, powerful, efficient, and sophisticated solvers can always be added later. If basic methods are a success, then there even more reason to consider bringing more sophisticated methods to bare on the problems of robust form parameter design.

**Interval Contractor**

Now we define the general interval contractor using the definition given in reference [172].

**Definition 8.1.1.** Let $c$ be a constraint on the reals. A contractor for $c$ is a function $C : \mathbb{R}^n \to \mathbb{R}^n$ such that:

$$\forall b \in \mathbb{R}^n \left\{ \begin{array}{l}
    C(b) \subseteq b \quad \text{contractance property. and} \\
    \forall x \in b \not\in C(b) \quad c(x) \text{ is not satisfied}
\end{array} \right.$$

Here we see the fundamental properties of an interval contractor:

- Contractors can only ever contract the interval domain.
- Any subset of the interval range excluded by an interval contractor is guaranteed to be infeasible.
Furthermore, we should note that there are a set of properties that can be proven about specific contractors which can then be used to prove interesting things about algorithms using them. A contractor, $C$ may have some of the following properties:

- **monotonicity**: a contractor is monotonic if $x \subset y \implies C(x) \subset C(y)$

- **minimal**: a contractor is minimal if $\forall$ interval box $x$, $C(x) = \Box A = x \cap X$ and again, $\Box A$ is the interval hull of the set $A$.

- **thin**: if for all points $x$, $C(x_{\text{thin}}) = x_{\text{thin}} \cap X$. A thin interval $x_{\text{thin}}$ is an interval which has degenerated to a single point.

- **idempotent**: A contractor is idempotent if for all boxes $\Box x$ we have $C \circ \Box x = C(\Box x)$. That is, further application of the contractor operator yeilds the starting interval. The contractor is thus a map from the interval domain the interval domain. Thus $x$ is a fixed point relative to that contractor.

- **convergent**: a contractor is convergent if for all sequences $\Box x(k)$ of boxes containing $x$ we have $\Box x(k) \to x \implies C(\Box x(k)) \to x_{\text{thin}} \cap X$

We will not prove that these properties hold for our particular interval problems, but rely on the reader to consult the interval literature for more information regarding the particulars of interval mathematics. Note though, that we will rely on these properties from time to time. For instance, the convergence property ensures that our algorithms terminate. Also, note that no attempt has been made here to analyze the computational complexity of the sub-problems attacked here in this ship hull design package.

Finally, though a contractor will always rule out infeasible portions of an interval, it cannot generally reduce an interval to only it’s feasible domain in all cases, even iteratively. In practice, though, we can often make useful progress. That is all we need to eliminate portions of the infeasible domain.

Continuing with the general outline of contractors, they typically come in two forms:

1. Iterative, or Newton-like “box consistency” contractors

2. Non-iterative, “hull-consistency” contractors

Now we need to specialize to interval hull consistency, which is the only type of contractor made use of in this thesis.
Hull Consistency

Hull consistency (HC) is the basic algorithm which will enables constraint propagation to actually do work on our constraints — to enforce them on our design space. Our hull consistency is a local contractor acting on a single constraint equation, or set of equations, which by inversion acts to reduce the width of the interval domain of the constraint. \(^5\) For clarity we focus on the version which acts on a single interval expression. See the interval optimization literature, e.g., [6], for details. The essential point is that the algorithm involves solving for particular terms in the expression, and intersecting the result with the initial value for each particular term.

Consider an equation \(f(x, y)\), defined to be a constraint with interval inputs and \(f(x, y) = 0\). We can say that values of \(x \in x\) and \(y \in y\) are consistent relative to the function \(f\) if for any \(x \in x\) there exists some \(y \in y\) such that \(f(x, y) = 0\) and for any \(y \in y\) there exists some \(x \in x\) such that again, \(f(x, y) = 0\). The notion is general to any number of inputs. Obviously, then, a contractor can be written to enforce each consistency relation. In our program this will be performed on expression in a systematic and automated way.

Computing Hull Consistency

We compute local hull consistency by inverting (solving) a single constraint directly for each term in the constraint which is also an interval variable of our design problem. (This means that each term is some kind of form parameter in our design problem.) This notion of consistency is based on the global consistency of every isolated constraint \([172]\). Thus making HC “global” over the design space, since it involves the entire interval, but also “local” to that particular constraint only. \(^6\)

In practice, the constraint in question should be computed “forward” and “backward” to fully propagate any domain reducing information contained anywhere within the equation \([173]\). This is because additional narrowing may occur when computing a “nearby” contractor for another subexpression. The subsequent reduction in the intervals might mean that a previously computed component can now be narrowed further. The forward, backward algorithms take care of this. Said a different way, in the case of a compound constraint,

\(^5\)The interval literature describes this as contracting an interval “box,” which in our case would be thought of as a subspace of the design space under consideration.

\(^6\)For those interested in more details of interval analysis, it might be fun to consider the duality of local and global efficacy here between “local” hull consistency and “global” Newton’s method: Hull consistency, which uses only local information, is known to be “good at” reducing the size of large interval boxes. Meanwhile interval Newton’s method, especially when incorporating all equations of a nonlinear problem, is fully global in information content, and yet is is known to be “good at” rapidly converging small boxes to tolerance, and for having difficulty reducing large domains. Thus the method incorporating global information is powerful locally, and the method which uses more localized information is better at large scale reductions in the problem space.
the equation must be fully processed so that each component in turn receives all information contained in the remainder of the expression. This necessarily complicates the implementation of hull contractors, especially in procedural programming. Note also, that if each variable in the equation occurs only once, then the forward backward algorithm is “minimal” in the sense defined above in section 8.1. Lastly, the resulting operation, even if minimal, may not produce sharp bounds on the components of the constraint being contracted by the operation. In practice, though, some bound will be computed, and (jumping ahead a bit) a single expression may in turn be part of a larger network of such constraints. Iteration across all constraints may yet allow for further bounds reduction.

The interval hull of a set $S$ is denoted by $\square S$, and the interval evaluation of an expression $f$ is $f(x)$, so that $\text{range}(f, x) \subseteq f(x)$. The interval hull is the smallest box which enclosed the set $S$ or the enclosure of our function so that $f(x)$ is the enclosure of $\text{range}(f, x)$. Our hull consistency algorithm will produce a rigorous (outer) bounds of the hull of our constraint expression, hence the name. 

As stated above, hull consistency is derived from to inverting the constraint directly. With this simple idea as background, differing ways of implementing hull consistency could be imagined.

In this section, we keep things simple and focus more narrowly on computing the hull consistency of a single expression. Here a particular interval mathematical expression is evaluated and inverted, propagating the data contained in one interval to the other intervals related by the expression. In general though, the constraint should be computed “forward” and “backward” to fully propagate any domain reducing information contained in a compound equation, as mentioned above. Further, we may need to iterate in some fashion so that each component in turn receives all information contained in the remainder of the expression. Otherwise, there will be some portion of the domains of each component which are not as sharp as possible. Even for a fully propagated constraint, sharp bounds will not result without iteration unless each variable present in the constraint appears only once. Moore, et al. [102], have a good introduction to the particulars of these interval iterative techniques.

One particularly important property that an interval hull consistency algorithm must maintain is for the domain of each component interval to be closed under update. That is, interval updates will always take place using the interval intersection operation. This will ensure that we consider only solutions found within the original design space, and that our intervals do not grow without bound.

---

7We pass briefly over the details of correctly stating definitions for interval quantities here. See Emden, et al. [174, 175] for a more in depth discussion.
Form Parameter Hull Consistency  For ship hull design, a form parameter constraint might take on something as simple as the following form:

\[ f(x) = x - h(x) = 0 \]

“Hull” consistency then demands nothing more than, in principle,

\[ x = h(x) \]

where portions of the domain of \( x \) will be trimmed away when they do not contain solutions to the constraint. But of course we enforce hull consistency by solving for \( x \) and update by intersection: \(^8\)

\[ x_1 = x_0 \cap h(x_0) \]

Consider a situation where some computation necessitates the propagation of a simple binary relation where \( a \) and \( b \) are intervals \( a = [x_1, \bar{x}_1] \) and \( b = [x_2, \bar{x}_2] \). Assume the relation is,

\[ a = b \]

Now let us suppose that some outside computation produces the following values:

\[ a = [0.5, 3.5] \]

\[ b = [1.5, 4.5] \]

Hull consistency proceeds in two steps. In the forward step, the information in \( a \) is propagated directly to \( b \) since the relation is a simple equivalence.

\[ b = b \cap a \]

\[ b = [1.5, 4.5] \cap [0.5, 3.5] = [1.5, 3.5] \]

\(^8\)Here already we see proto-iteration at work.
Thus $b$ has been contracted, or narrowed. In this example, we have used the above relationship to filter out an infeasible portion of $b$’s starting interval.

The final step in this hull consistency algorithm is the backward propagation of the information. Since $b$ has already been narrowed, the algorithm would do well to take advantage of this information. Accordingly, the procedure for $a$ is simply

$$a = a \cap b = [0.5, 3.5] \cap [1.5, 3.5] = [1.5, 3.5]$$

The hull contraction operation (filtering) is now complete for the information contained locally in this term.

More complicated constraints will follow the same procedure, but, the forward and backward phases of the computation must be extended to fully propagate the information across all components. When this is carried out in traditional procedural programming, these extended operations must be hand coded. For example, consider the constraint

$$a + b = c$$

Full propagation requires we compute

$$a = a \cap (c - b)$$

$$b = b \cap (c - a)$$

$$c = c \cap (a + b)$$

If any of the constraint components have changed during the forward and backward steps, we run the algorithm again. We do this because computationally, at least with a naive implementation, we do not know whether some variables in the algorithm are the same, and thus we do not know how much effected the computations might be by the dependency problem - a particular issue that will be detailed in subsection 8.1.

So iteration can fruitfully proceed as long as it makes progress in narrowing at least one of the components, or until they are narrowed sufficiently as governed by a tolerance. Note that for such a simple constraint, one pass through the algorithm will be idempotent, i.e. it will reach a fixed point. In general many iterations may be required. 

Finally, there is also the possibility that even the very first pass of the forward backward iteration narrows none of the terms involved. Then iteration would most fruitfully be terminated at that point.

---

9This procedure is known as substitution iteration in the interval literature [103]. Here it is basically constraint propagation across the individual components of one constraint expression.
Algorithms: Hull Consistency

Note that the decomposition method of programming constraints has been shown equivalent to a computational graph parsing approach similar in feel to reverse mode automatic differentiation [86, 173]. In those papers, if the later approach is supplemented with techniques to avoid full decomposition then it is known as the HC4 algorithm. Otherwise, in general, sub-terms may need to be assigned “dummy variables” to store intermediary data.\textsuperscript{10} The HC4 algorithm adds extra data to the computational graph to store the \textit{fwd} and \textit{backward} pass interval data at every node. In this way it avoids an explicit expression decomposition step. On the other hand the basic constraint decomposition approach is known as HC3. We prefer to call the algorithm arc consistency, as this may be the more general term in literature. This algorithm processes the equivalent information, but each node in the computational tree is decomposed such that a dummy variable is used to store interval data resulting from subcomputations in the case of n-ary data. Thus the HC4 algorithm is a bit easier to visualize, as it naturally has less clutter from dummy variables. Also it is known to be more efficient [173] in some cases. Otherwise they are similar.

Note that in either method the computational graph for one mathematical expression, (actually a syntactic tree — a simpler data structure), is constructed and explicitly traversed in both forward and backward directions, propagating the interval results to all variables until a fixed point of the iteration, or some close tolerance on the amount of change per iteration, is reached.\textsuperscript{11} In this thesis, we stick with the arc consistency, “AC3,” method since it is very general. As we shall show in section 8.2, we process the computational graph to automatically generate the extra connectivity, or, “dummy variables,” needed to handle intermediate (sub-expression) data.

Regardless of which algorithm is used, for general \textit{n-ary} constraint relations, iteration is required to fully propagate all information. Again, iterative termination is determined by either convergence to an interval value of tiny, predefined width, or if the interval data stop narrowing. In practice then, we use the interval convergence rule given in subsection 8.1, modified to have some tolerance, since we do not need the absolutely converged solutions for our engineering purpose here.

\textsuperscript{10}Looking ahead, one might imagine using operator overloading together with the built-in parser of a programming language to produce such sub-terms, and along with them, who arithmetic expression trees. This is where we are heading.

\textsuperscript{11}A single mathematical relationship is represented by a syntactic tree. When we move to encompass the relationships between rules, in full constraint logic programming, each individual tree will be tied together by usage of the same variable in other trees, forming a general undirected graph.
As an example of the full forward backward algorithm, we examine the interval equation

\[(x_1 + x_2) \times x_3 \in [1., 2.]\]

In this equation, we assume each interval is unique — and not dependent on any of the others, except as shown in this expression. First, the equation is parsed into a computational syntax tree, as shown in figure 8.1. Operator overloading allows Python to generate such trees by amending variables to store data about which inputs and operations are used to create them. By the same process, a set of dummy variables are created to store “partial” information. This is data computed from sub-terms and relationships within the mathematical expression. These dummy variables are denoted \(a\) and \(b\) in figure 8.2. These are also referred to as “intermediate terms.” Now we are ready to compute. First we compute the forward pass in figure 8.3. Data propagates from the terminal nodes (intervals \(x_1, x_2, x_3\)) to the dummy variables, \(a\) and \(b\), and finally to comparison of \(b\) to \([1., 2.]\) at the root of the tree. Intermediate data is always stored for future use.

Lastly, figure 8.4 shows backward propagation, which finishes the work of the algorithm. Since each
Forward-Backward Contractor

$(x_1 + x_2) \times (x_3) \in [1., 2.]$

Intermediates
Terms:
\[ a = x_1 + x_2 \]
\[ b = a \times x_3 \]
\[ b = b \in [1., 2.] \]

Figure 8.2: Creation of intermediate terms, $a$ and $b$, which will allow interval values to propagate up the syntax tree.
Figure 8.3: Forward propagation. The algorithm carries out the basic computations of the rule, contracting the interval widths where possible.
Figure 8.4: Annotation tree for the backward propagation of the constraint. Dummy variables again aid in propagating data — this time back down the tree to the terminal interval nodes, $x_1$, $x_2$, $x_3$. 
interval is independent of the others, no further computations will narrow the intervals used in this equation. We could check this by running the forward pass. In practice, our algorithm will terminate when no further contraction takes place. This allows for early termination when data itself precludes further contraction.

As alluded to above, in our system intermediate nodes are naturally and automatically created by the Python interpreter. The details of this programming construction will be shown in subsequent sections. Python will use dummy variables to capture this data (and our system will track these dummy variables and ensure they remain unique). This can be an important detail for declarative logic programming.

An Extension: Linear Combinations of Functions, e.g. Line Constraints

We see above that our program will apply H.C. to each equation of a system of constraints in order to try and eliminate parts of the box that cannot contain the solution. Alternatively, we can sometimes eliminate larger parts of the box if we apply H.C. to a linear combination of equations from the system.

By adding linear combinations of one constraint to another constraint, we rotate and stretch the constraint equations so that they coincide with the coordinate axis. This might be an interesting aspect to consider for future versions of the system.

The overall point in the above discussion is to understand that intervals allow powerful inference about even global nonlinear optimization problems. Further, local contractors allow algorithms to propagate detailed local information when using global interval contractor methods. This allows a program to compute critical narrowing operations on the feasible bounds of individual design parameters. As demonstrated with the basic hull consistency examples, the interval arithmetic calls out for algorithms that can process constraints gracefully. This means allowing information to propagate forwards and backwards through a generally complex system of numerical relations.

Interval Arithmetic with Division by Zero Finally, during a computation, an interval containing zero could well be found in the divisor of a division relation. Fortunately, a generalized, or extended, interval arithmetic has been developed to handle such a situation, as is noted in global optimization texts using intervals, such as that of Hansen and Walster [6], Kearfott [103]. Such an extended interval division was originally purposed by Kahan [176] for continued fraction research, and also independently by R.J. Hansen [114] as noted by E.R. Hansen [177]. Further, note that Novoa [178] and Ratz [179] later proposed corrections that make generalized interval division consistent for use in optimization and nonlinear equation
contexts. Since we are discussing division by zero, it should be stressed that all interval updates proceed via the intersection operation, and this ensures interval form parameters will always remain bounded. Indeed, they always remain within their original bounds, irrespective of any update, division by zero, or wrapping effect.

This is not the end of the story though. Interval division by zero can return a splitting into two intervals, which when unified (intersected) with their originator, split the original interval domain into two disjoint interval sets. This would present another difficulty to traditional, procedural programming techniques, as the systems would need to be able to handle multi-valued returns in general.

In practice, some interval practitioners avoid this situation by returning the hull of the two intervals. Nevertheless, we would like to be able to incorporate the split intervals directly. The reason we would like to keep the two-valued singular division return capability is that the resulting intervals split around problematic areas of a design space. They effectively sense and remove portions of a design parameter which were sending a computation to infinity. Generalized interval arithmetic can be said to precisely locate and excise singular portions of a design space. This is a fantastic property, well worth keeping if possible. It should be noted however, that alternatives exist. Interval paving or interval branch and bound algorithms which take the hull of two valued division expend extra computation to find and split out such singular portions by heuristic design space splitting. This alternative, in addition to being less efficient in higher dimensions, also necessitates the embedding of the local optimization problem into such a paving or branch and bound strategy — better suited to global optimization approaches.

We will show that relational programming, especially in languages like miniKanren, already comes equipped to handle such potentially onerous objects as the two valued, or even multi-valued function return [152, 154]. Furthermore, in trials the system to be developed here was first tested with simplified (outer hull taking) division which remained single valued. The system was found to be unable to make progress on many constraint (design) spaces. When the algorithms were updated to take advantage of interval splitting, progress was suddenly made, and convergence achieved. Perhaps this is not so surprising, given that the literature describes extra computations to work-around the problem. Doubtless we could have incorporated such methods as well. Here though, thanks to declarative logic programming, it was simpler to simply allow the design space to split as a matter of course. Therefore, we now show the details of generalized interval division.
Algorithm: Generalized Interval Division

The standard rules of interval arithmetic preclude division by an interval containing zero. In formulating the system for feasible form parameter design, removing this restriction proved useful. Infinite interval division was discussed in work by R. Hansen [114] and prior to that for special kinds of arithmetic by Kahan [176]. The interval community thus generally attributes the invention of the technique to Kahan. “Extended,” or infinite interval division extends the traditional real interval arithmetic by allowing division by an interval containing zero. An immediate consequence is that extended interval arithmetic can be an algebraically closed system. The definition for generalized interval arithmetic is given below.

Given an interval quotient \( \frac{x}{y} \), where \( x = [a, b] \) and \( y = [c, d] \) generalized division is as follows:

\[
\frac{x}{y} = \begin{cases} 
[b/c, +\infty] & \text{if } b \leq 0 \text{ and } d = 0 \\
[-\infty, b/d] \cup [b/c, +\infty] & \text{if } b \leq 0 \text{ and } c \leq 0 \leq d \\
[-\infty, b/d] & \text{if } b \leq 0 \text{ and } c = 0 \\
[-\infty, +\infty] & \text{if } a < 0 < b \\
[-\infty, a/c] & \text{if } a \geq 0 \text{ and } d = 0 \\
[-\infty, a/c] \cup [a/d, +\infty] & \text{if } a \geq 0 \text{ and } c < 0 < d \\
[a/d, +\infty] & \text{if } a \geq 0 \text{ and } c = 0 
\end{cases}
\] (8.1)

Generalized interval division as defined above will directly contribute to the robustness of the feasible form parameter inference system, as an example will show.

- division by an interval containing zero acts as a sensor which detects singular portions of the design space.
- Then generalized division can be used to excise singular portions of the design space.
- automatically splitting interval design parameters to return de-singularized spaces to the list of possible search areas.

The cases above can be divided into those where \( b \leq 0 \) and those where \( a \geq 0 \). If zero is on the interior of the

\[12\] This is separate from the closure property seen in using interval contractors, where the space reachable by contractor is closed by construction.
upper interval, then no new information can be gleaned from processing the interval. In practice all the cases above occur frequently. To reiterate the importance here, note that in programming systems which are not able to accept lists as return values, the consistent alternative is to always return the full domain, $[-\infty, +\infty]$ for any division which would be split. This comes at the cost of the singularity sensing and removing capability of the interval unit system. In developing the feasible form parameter code, first a version was tried which made such a concession to ease of development. It was unable to successfully narrow the design domain when initialized with ‘wide’ starting intervals for the form parameters. This shortcoming was rectified by using the full list processing capabilities of miniKanren, to be developed in detail in subsequent sections.

The Dependency Issue

The so-called dependency problem is the primary drawback of interval operations on computer. In the examples shown above, we’ve alluded to this problem but now we give a more direct description. The problem is simple: The wider the starting interval, then in general, the wider the resulting interval from some computation of an interval function or expression making use of it, and critically, the greater the overestimation of the resulting range. This overestimation is “interval dependency.” Furthermore, the form of the expression can induce widening as well. Although interval methods can determine elementary arithmetic operations to (outward-rounded) machine precision, this is not generally the case with more complicated functions. If an interval occurs several times in a calculation, and each occurrence is taken independently by the interval system, there will be a widening of the results. A simple version of the problem can be seen with subtraction. Following Hansen, [6], subtract an interval $X$ from itself.

$$[a, b] - [a, b] = [a - b, b - a]$$

New users of interval arithmetic might have hoped for a result of $[0, 0]$ but that would only follow if $a = b$ exactly. Instead, the result conforms to

$$x - y = \{x - y \mid x \in x, y \in y\}$$

rather than exact negation, $x - x = \{x - x \mid x \in x\}$.

For similar reasons, it is important to express powers as they are, rather than say, perhaps following a
practice taught for efficiency reasons to C programmers, re-writing them as a term multiplied by itself a certain number of times. This can be seen straight from the definitions of multiplication and power.

Multiplication is given by

\[ x \times x = \{ x \times y \mid x \in x, y \in y \} \]

Where, for example, the power of two is defined as,

\[ x^2 = \{ x^2 \mid x \in x \} \]

Clearly, the (interval) power function can out perform (interval) multiplication of the same terms.

The issue extends generally such that each occurrence of a variable is treated independently of any other occurrence of the same. Thus, a sub-optimal enclosure will be formed in evaluating an interval expression of the form \( \frac{x-y}{x+y} \) but this could be optimized by writing it in the form \( 1 - \frac{2}{1+y} \), assuming no division by zero occurs. Some standard interval algorithms are often written with special cases that check for multiple occurrences of the same variable, but in general a widening of the resultant intervals is to be expected relative to what a perfect result would give. See Hanson “dependent subtraction” [6], also called “cancellation,” for the version of the subtraction algorithm which avoids the dependency problem shown above.

As might be guessed, general attempts to reduce this problem by modifying expression form involve term re-writing techniques. Following the logic above, one could imagine building a term re-writing system to generate the optimal form for a given expression to be evaluated over the intervals. Such systems based off of general term re-writing methods [180, 181], extended for interval analysis, have been shown before [182]. Indeed, the utility of such systems has been noted before. Kearfott [103] notes that “…since interval computations benefit from a priori rearrangement and simplification of expressions, and since algorithms such as interval Newton methods require evaluation of functions and derivatives with different data types, it is natural to consider interval computations within a symbolic computation environment.” Though he also notes the drawback in terms of performance!

Later on in this thesis we will be compiling mathematical expressions into logical rules. We could imagine re-writing out terms into more optimal form during that process. But in this thesis our individual expressions are closely intertwined with other expression via re-use of the same terms. It would seem, then, that term re-writing is closely related with equation solving and contractor programming. There is a computational
trade-off to find the optimal balance between searching for the best expression of the terms, and solving the
equations simultaneously. Indeed, term rewriting literature contains numerous complexity studies. Some
recent studies prove worst case bounds and decidability in shallow cases [183], special cases extended to
more general systems for inferring upper bounds [184], and automated procedures to find bounds on the
complexity [185, 186]. We will not go into detail here.

Furthermore, term re-writing can be said to be generalized by “narrowing” [180], where the term
matching process is replaced by unification. Interestingly, we will see that unification and narrowing, as
interval contraction, is exactly how we enforce consistency across our initial system of design constraints.

It seems that term-re-writing could still play a role in local optimization of the interval computations.
This could be the subject of future work in optimizing the code base. Furthermore, other interval dependency
reducing techniques exist that could also help address the dependency issue. For example, centered forms,
monotonicity exploitation [169, 172, 187], Taylor model (polynomial) reduction techniques [188], the
aforementioned techniques utilizing combinations of constraints, and of course, interval splitting. (Interval
splitting is the simplest to describe. Since interval dependency is worse for wide input intervals, simply
splitting an interval into pieces and evaluating them separately can be a particularly effective technique!)
Some of these could be drawn from in order to build up the effectiveness of the constraint solver, though each
may be more adapted for particular purposes. Monotonicity, for instance, is particularly suited to situations
where derivatives are available, and branch and bound can also be utilized in supporting interval generalized
division.

Note that some widening is still to be expected in general interval computations, no matter how cleverly
coded, as floating-point arithmetic is only approximate, and so any interval system must “round outwardly”
in order to bound the true result from outside. But please note that widening resulting from outward rounding
is useful, as opposed to the widening described above! This technique is a requirement for any rigorous
computations, but may not be as critical to our purposes, since we merely want to find consistent solutions
at random in the design space, and do not try to find all such solutions with complete confidence. Outward
rounding does, however, protect us from ill conditioned floating-point operations. Here we might be willing to
trade accuracy for speed, nevertheless we include outward rounding in our implementation since it’s benefits
seem to come at minor cost, when implemented well.

In the end, the primary takeaway here is that the dependency problem is a general feature of interval
arithmetic and it is to be expected that somewhere in our interval computations a widening effect will occur.
One might worry that this precludes the use of intervals outright, as, for example, in an iterative procedure the dependency issue might be expected to grow an interval without bound. A scary thought, but fortunately we operate in a bounded design space, and our interval design variable update computations will take place by intersecting any resultant interval with its prior value. Unbounded growth of intervals thereby simply does not happen. Combined with contractor programming, dependency does not preclude the algorithms from narrowing spaces into solutions, proofs, and disproofs. Nevertheless, measured steps can be taken to reduce dependency for efficiency reasons.

**Ramifications for Form Parameter Picking**

We note also that since intervals are not perfect, i.e. they capture the feasible domain, plus some “extra width” in each interval, will have slight ramifications for our search algorithms developed later in this thesis. The issue is this: Due to the dependency problem, some portion of an interval might be infeasible, and yet stand un-reduced from that interval, and from our design space, even after our contractors have done all they can. We will see how this is handled once we get to the design selection phase of the program.

**Interval Arithmetic Implementation**

That was a brief high level view of the power of interval techniques which also motivated the need for programming approaches that may be different from the conventional forward operators and assignments of procedural programming. With this as motivation for pursuing them further, we now look at the implementation details.

Interval arithmetic is implemented by operator overloading in Python. The key to this is to efficiently use Python’s underlying C, IEEE 754 floating-point representation to accurately bound computations expressed with our overloaded operators. This is the IEEE supplied standard for rounding and various particulars of an interval arithmetic implementation [171, 189]. This has already been implemented by the pyInterval library, which for correct, fast, outward rounding is made available [118] via the FPU module. The numerical operators from pyInterval were not used, as again complete control of the implementation was helpful in integrating with the AD, B-spline, and CLP aspects of the program. That said, if this research code is upgraded to an industrial version, most likely the step taken would be to incorporate an existing and heavily tested library. At development time the chief python library for interval analysis was still under development and the chief concern was interoperability with the geometric and automatic differentiation classes. Thus a custom interval arithmetic class was implemented for this project. In trials, it was surmised that a switch to
that standard would change numerous other portions of the code base, due to the interaction between all the
components, and has not been incorporated into the code base for this study.

Note also that outward rounding is necessary for rigorous solution finding and computational existence
proofs, but is not actually necessary for constraint propagation itself, in the continuous, set based, but still
computationally approximate, sense. In our program, we assume that the user does not wish to rigorously find
all or some solutions, but instead wishes to quickly find solution vectors that are likely to work in practice.
Therefore, we suggest the possibility of ignoring the customary interval outward rounding necessary for true
theorem proving using interval computations, in the event that this is a helpful speed optimization trade-off.
Instead, we might eliminate this step if it were expensive and our chief wish was to generate valid, quality
designs as quickly as possible.

As it is, we shall make use of outward rounding, especially to avoid having to caveat all of our use of
interval techniques with warning labels. In practice, even with interval analysis written in Python software,
but using specialized rounding code from [118, 119], the algorithms were found to be “fast enough” for our
purposes. Actually, the primary factor in solution speed seemed to be the conditioning of the constraint
system itself. If there is a large set of reachable, feasible designs contained within the design space, a
particular solution will more quickly be found. Furthermore, if there is no need to bound the solution as
tightly as the hardware would allow, a trade off can be made to find feasible design sets to within some finite
interval tolerance which will also decrease iteration times. In practice, we pick a tolerance that is close to the
acceptance tolerance of the nonlinear solver, so as not to waste computation, but to be sufficiently sure that
dependency issues are not masking an infeasibility inherent in the design vector.

For our implementation, operator overloading proceeded as follows:
Source Code 8.1: Implementation of the interval analysis class “ia” with some basic operations shown

class ia(object):
    infinity = 1.e25
    def __init__(self, inf, sup, isempty = False):
        self.inf = inf
        self.sup = sup
        self.inf = fpu.down(df(self.inf))
        self.sup = fpu.up(df(self.sup))

    def __add__(self, other):
        if isinstance(other, ia):
            return ia(self.inf + other.inf,
                      self.sup + other.sup)
        else:
            return ia(self.inf + other,
                      self.sup + other)

    def __radd__(self, other):
        return self.__add__(other)

    def __sub__(self, other):
        if isinstance(other, ia):
            return ia(self.inf - other.sup,
                      self.sup - other.inf)
        else:
            return ia(self.inf - other,
                      self.sup - other)

    def __rsub__(self, other):
        return self.__sub__(other)

- The use of a large number for infinity is a concession to development speed. Future refinements of the program should follow the IEEE conventions for intervals.
- isempty is a flag denoting an empty interval. This is used to notify the logic programming system of empty intersections. Such an interval denotes in-feasibility and can be pruned from the list of design spaces.
- Implementation of an interval operation is a matter of including the necessary logic to find and deal with interaction with standard real valued types, which is handled by isinstance checks, as well
as implementing the needed arithmetic operations as described earlier. In the full dissertation full
implementations can be given in an appendix.

**Code Example**

Application of generalized division is illustrated with an example. We intend to solve a simple interval
relation such as \( C = \frac{A}{DNB} \) for \( C \). Note that in general the program might need to solve for any of the variables
in this expression. With only the tools of interval arithmetic available, the function would need to be manually
re-organized by a human programmer in order to solve for another component variable. This would present
issues later when incorporated into large automated constraint systems, but the example analysis shown here
will focus on the simple case when the terms are already in the correct form for solving the equation for
the variable on the left hand of the equation (the assignment side in a traditional imperative programming
language such as Python).

As an aside we must not fail to mention that traditional optimization provides ready tools for addressing
the re-write issue via traditional optimization algorithms, such as gradient descent or Newton type algorithms.
In the field of interval analysis these are known as box consistency algorithms, and they are very powerful
because they can take advantage of global constraint information, and act on any number of interval design
variables at once. The primary weakness of interval box consistency algorithms is the fact that the global
nature of the algorithm makes them more susceptible to interval widening by the wrapping effect, and since
the algorithm is processing more, or all, of the design space, the algorithm is more likely to get stuck due
to containing multiple solutions, which may be far apart, or, and especially, the program may be unable to
converge upon the solution due to the presence of singularities, if generalized interval division techniques
are not available. The other reason for computing hull consistency type algorithms such as this one for the
simple constraint given here are that they are simply cheaper, computationally speaking, since they require
no derivative information, and sometimes iteration can be avoided as well.

The code in Listing 8.2 performs the following operations:

- Each variable is instantiated as an object of type \( \text{IA} \).
  - The first value is always the lower extremum of the interval
  - The second value is always the upper extremum of the interval.
  - The interval represents all real values between and including these bounds.
- Intervals $a, b, c,$ and $d$ are instantiated to a set of values.

- The expression $\frac{a}{d \times b}$ is computed (in an optimization context).

Source Code 8.2: Instantiation of and Extended Division with IA Variables

```plaintext
#Initial Values:
\begin{align*}
  a &= \text{ia}(150., 250.) \\
  b &= \text{ia}(-5., 20.) \\
  c &= \text{ia}(0., 1.) \\
  d &= \text{ia}(10., 20.) \\
  c &= a / (b \times d)
\end{align*}
```

Equations (8.2a–8.2c) show what IA routines actually compute when tasked with evaluating the singular expression $c = \frac{a}{b \times d}$ in an optimization context.

\[
\begin{align*}
  c &= c \cap \frac{a}{b \times d} = \frac{\text{ia}(150., 250.)}{\text{ia}(-100., 400.)} \\
   &= \text{ia}(0., 1.) \cap [\text{ia}(-\infty, -1.5), \text{ia}(0.375, +\infty)] \\
   &= [\text{ia}(0.375, 1.0)]
\end{align*}
\]  

As seen here, despite the fact that the denominator of the quotient includes zero, the interval operation was still able to contract the interval $C$. This is because the implementation uses the generalized interval division formulas as shown in figure 8.1.

Interval results always bound machine precision from above and below. If each interval variable only appears once in a computation, then the computation is minimally enclosed, that is, the wrapping effect does not occur.

**Summary**

The interval implementation presented here supports operations needed to augment a basic form parameter B-Spline curve design system in such a way as to support more feasible, robust computational procedures. It is fully compatible without Automatic Differentiation (AD) system as well, and in such use the interval
type is the primitive type with AD being the higher level controller, ensuring all derivatives and values are computed using interval methods.

We are now ready to see how interval analysis is used in relational constraint solving over the real interval form parameters. This “constraint logic programming” is the subject of the next subsection, 8.2. As demonstrated above, our hull consistency algorithms require forward/backward evaluation of constraints and extended interval division requires handling a multi-valued return. A change in programming paradigm will be most helpful in handling computation that can go forwards and backwards, and functions that can return more than one value. As we will see, unification and arc consistency over undirected graphs will handle the directional data flow issues, while the data structures of declarative logic computing will naturally accommodate the multi-valued nature of results. Together, these features of declarative and constraint logic programming make it the ideal compliment to interval analysis for this use case.

**Constraint Logic Programming Scheme**

Examine for a moment a simple the mathematical expression,

\[ x + y = z \]

In the traditional procedural programming context we have an expression which combines two variables, \( x \) and \( y \) to get a third, \( z \). In a traditional programming context the idea expressed above must become an instruction, or recipe, explaining to the compiler how to combine two inputs to get one output. Note that if this is indeed the desired recipe, we have written this statement incorrectly. We might expect a compiler error unless we restate this as an assignment:

\[ z = x + y \]

Furthermore, we must ensure that each component of this expression evaluates to something that is well-defined in our language, or further errors may result. That is, \( z \), \( y \), and \( x \) must already be assigned values before they are combined as above. These are standard realities in procedural programming, but there are other computing conventions which do not abide by these rules. Relational programming dispenses with this paradigm and its peculiar issues by eliminating the idea that inputs are different from outputs in the first place.
With relational programming, starting from the expression, instead of requiring the programmer to write down a statement or recipe conforming to the requirements of the machine, we will instead have a relation which expresses the programmer’s wishes. This relation acts as a constraint, and it is the computer’s job to derive the consequences, or, in other words, to return states in which this constraint is valid.

In practice, compared with procedural programming, the same underlying mathematics is represented with a different kind of flexibility. The relation makes no assumption about which direction data should flow. Instead, data will flow naturally to enforce the rule.

Relational databases follow the same kind of abstraction as found in relational programming. For instance, SQL is a language that allows the user to query a relation, but with a relational database the “implementation” of the rules and relations are typically specified as data tables. Different queries of the same table can treat different columns of that table as being inputs or outputs depending on what kind of question is being asked.

A relational programming language is like a relational database except that instead of a finite table of data stored on a disk, relations can be defined by a programmatic rule. The rule is similar to a function in a functional language as well. In functional programming, a function is a black box that has some inputs and an output. One “runs” the function by providing some input values and then receiving some output upon execution of the function. This concept of a function is a special case of a more general idea called a relation. A relation has a rule within, and that rule may have the same expression as the function of a functional language, but the rule is now a constraint which can be queried. We can query a relation to ask, “given some initial conditions, what does the constraint imply?” For example, if we have a constraint, \( x + 2.0 = y \) and a state such that \( y = 3.0 \) and \( x \) might be undefined (or have some value), the relation implies that \( x = 1 \). If \( x \) had been something inconsistent with this, e.g., \( x = 10.0 \), then the relation would have returned the empty set, since no satisfactory, or consistent, value exists to satisfy the given constraints. Incidentally, this simple example alludes to the utility of interval arithmetic, for we could imagine that a computer will have a much easier time satisfying relations over continuous sets, than floating-points.

Next we will explain exactly how relational rules are constructed and show how they are used. The components that make declarative rules programming possible will form the underlying, lowest level, methods of the constraint logic programming system or language. The basic building blocks are:

- Variables and values (any programming language will have these)
- Environment (state — a map from variables to values)
• Procedures (relational rules, also known as “goals”)

• Evaluation (unification — converting rules into updates to the state of the system)

In the paragraphs below, we explain these components and give a basic implementation of them. This will serve as the basic building blocks for a ship hull rules system.

Goals

Goals are functions or procedures which contain rules that we would ask our design space to maintain. Once a goal is built, a programmer, or program, can ask the computer to pursue states in which that goal is valid to the exclusion of states in which it is invalid.

We would like for the computations to return a list of states in which the goal is valid, or return a null state in the case where the rule cannot be upheld. To build up goals we first need to talk about variables and the states, or environments, in which those variables map to values. Only then can we talk about procedures (goals) and some kind of evaluation process for the language. This discussion is modeled after similar discussions building up the domain specific constraint logic programming language miniKanren [152, 154].

Variables

In logic and relational programming languages, variables are first class entities. This means that unbound variables live separately from their values and can be passed around like any other first class data in a particular language, such as bound variables or functions in Python. The Python implementation of the variables class is given next.
class Variable(object):
    ""
    sqKanren logic variable class
    These are the objects which will be combined in relations
    -the geometric logic programming rules used
    to enforce feasible form parametric design.
    ""
    When passed into a sqKanren environment,
    they map to other data, typically
    either interval values
    or simply the name of the variable itself,
    if data is lacking.
    ""
    def __init__(self, name='', nbrs=None,
                 arcs=None, dummy=False, flist=None):
        self.name = name
        self.dummy = dummy
        if name == '_':
            self.dummy = True
        if arcs is None:
            self.arcs = []
        else:
            self.arcs = arcs
        if flist is None:
            self.flist = []
        else:
            self.flist = flist
    def __repr__(self):
        return str(self.name)

In listing 8.3, the Variable class is created. For basic logic programming, the only real requirement for
this class is that individual variables can be distinguished despite having the same name. This is implemented
via pointer equality. That is, equality means that two variables point to the same object. Names are thus
mere place-holders for convenience. Distinct objects that have the same name are not considered equal.
Other inputs to the __init__ method shown in the class above will primarily be needed once we extend the
relational language to be a full constraint logic system for general graphs of rules.
State

State is a data structure which associates variables with values. Our states class will extend the python dictionary. The dictionary will map from logic Variables to interval values represented by the ia class. These maps will in turn be encapsulated in a list. This list of State classes will be known as our States class.

A single state object will be equipped with some special methods to facilitate logic programming. These operators are then extended to list processing in the States class. The actual code-base used was originally derived from miniKanren, and thus tends to look like miniKanren in its particulars, some of which are completely necessary to our purposes, and some of which are not. Here we focus on only those methods which are strictly necessary. If any methods are mentioned which are superfluous to our usage in this thesis, this will be noted when they are discussed.

Source Code 8.4: States: A type of Python dictionary which maps logic variables to values. Following the design pattern of miniKanren, we equip our state with methods to bind variables to state, and to assign values to those variables. The practice in logic programming is to return a new state when binding and assigning new variables.

```python
class State(object):
    def __init__(self, values=None):
        self.values = values or {}

    def bind(self, name):
        """bind a name to state with no assignment""
        values = copy.copy(self.values)
        var = Variable(name)
        values[var] = None
        return (State(values), var)

    def assign(self, var, value):
        values = copy.copy(self.values)
        values[var] = value
        return State(values)

    def assign_bind(self, var, value):
        s, var = self.bind(var)
        return s.assign(var, value)
```

Here we summarize the important properties of the State class, and its extension to States, which we do
not give explicitly.

- In our logic language, a state is implemented as a python dictionary which maps logic variables to values.

- Logic variables will either evaluate to themselves, to other logic variables they have been unified with, or to intervals.

- A design space is a list of states where each state has a dictionary containing all the same design variables as every other state. These are contained in another class called States, which acts as a container for the list of states, and similar overloading to allow for the arithmetic operations.

- A particular State is unique, differing by at least one interval component from every other state. This facilitates the branch and bound paradigm, however we do not make use of this capability in this thesis.

We primarily process states using this list of states capability to allow for multi-valued interval division by zero. This capability is well documented in the interval literature as allowing a system to effectively divide by an interval containing zero. Furthermore, such generalized division, when used as part of an interval contractor algorithm, e.g. hull consistency, can still make progress narrowing towards a solution even in the presence of a singularity. See section 8.1 on interval analysis for details.

Source Code 8.5: state method, value_of looks up the value of a variable in an environment.

```python
def value_of(self, var, value):
    """miniKanren Walk""
    if key in self.values:
        value = self.values[key]
        if value is None:
            return key
        else:
            return self.value_of(value)
    else:
        return key
```

In code listing 8.4 the values attribute is a actually a Python dictionary. Future implementations might choose to subclass (inherit from) the Python dictionary itself, rather than making the dictionary an attribute as shown in this implementation. The simple version shown here avoids the possibility of complicating the
dictionary implementation while making data access slightly more complicated. Given that we only want
to access the dictionary using the methods of our logic programming language, this simple implementation
is sufficient. It is important to note that in general a value could be anything that the underlying language,
Python, could compare for object-equality. Equality comparisons are fundamental to this kind of logic
programming.

A state is an immutable object in logic programming, but we must be able to declare new variables in our
state, and to assign values to our variables in a given state. In listing 8.5 we give a method for our states class
which looks up the value of a variable in that state. We need this because variables can be assigned to other
variables. With “value_of” we always walk to the end of the chain of association.

To declare new variables in our state, we could follow miniKanren and have a function to both create
variables and bind them to values at the same time, or we could decompose these operations. The assign,
bind, and assign_bind methods handle these options. We could follow miniKanren and add a fresh method
to include computation along with the creation and binding of variables, but we do not need to do so as our
language is to be an internal component of our constraint programming system, and so we do not need to
provide this type of functionality. In practice, it is a choice for the language designer, but any such method
might be appropriate for our purposes. We simply need to create variables and bind them to values in some
particular state. These methods suffice.

Now that we have both values and state we should note that in this language “unbound” logic variables
evaluate to themselves. They may also evaluate to other logic variables or to intervals. Next we introduce the
algorithm which essentially transforms binding into a logical method upon which the constraint solver will
rest.

Unification

Unification is the process of making two values equal given a particular state by adding assignment statements
to the state in cases where two values are equal. Unification is a form of advanced pattern matching
[121, 150, 152]. Credit for the algorithm performing unification of first order logic goes to Robinson, [190].
Here in code listing 8.6, we extend basic unification to cover python lists and especially our interval data
type, the “ia” class.

Unification is the fundamental algorithm which makes relational rule computation possible, and indeed
forms the heart of the declarative, relational, constraint logic programming system. In most basic form, it only
works on binary equality relations. It might be worth noting that we did experiment with extending unification
to handle interval inequalities, but in practice these could be re-written as interval equality constraints. For
instance, a stipulation that $x \leq 1.$ could be implemented by defining $x = [\infty, 1.]$. Other extensions, such
as to support unification over lists, were not needed for this project. These are commonly used in, e.g.,
miniKanren documentation, to demonstrate relational programming techniques such as “running append in
reverse” — a precursor to relational programming in more general contexts such as relational interpreters for
code generation [156]. We attempt no such feats here. After we show the interval unification algorithm in
code listing 8.6, we will use this to create simple binary goals.
def unify(self, a, b):
    """miniKanren-like Unification with Support for the interval type: 'ia'
    """
    ivars = a, b
    a = self.value_of(a)
    b = self.value_of(b)
    if a == b:
        return self
    elif isinstance(a, ia) and isinstance(b, ia):
        values = copy.copy(self.values)
        test = a & b
        if test.isempty:
            return None
        values[ivars[0]] = test
        values[ivars[1]] = values[ivars[0]] & b
        values[ivars[0]] = values[ivars[0]] & values[ivars[1]]
        return State(values)
    elif isinstance(a, ia) and isinstance(b, Variable):
        values = copy.copy(self.values)
        values[ivars[1]] = a
        return State(values)
    elif isinstance(b, ia) and isinstance(a, Variable):
        values = copy.copy(self.values)
        values[ivars[0]] = b
        return State(values)
    elif isinstance(a, Variable):  # simple varlookup
        return self.assign(a, b)
    elif isinstance(b, Variable):  # simple varlookup
        return self.assign(b, a)
    else:
        return None

Notes about the algorithm

- a and b are logic variables

- value_of is a recursive function like miniKanren’s walk which returns the value of a logic variable in some state.

- ia denotes the interval class.
• **self.values** is an internal name for state.

  - Unify and other logic operations and rules are methods of the state and or *States* classes. *(States just holds a list of states where each *state* is an actual dictionary of design variables to intervals)*

  - Arithmetic operators are implemented as overloaded methods in the *States* class.

We will close this section with an example of usage of the relational capabilities of this language by developing a simple goal example, and in the next subsection, examine the kinds of rules we will need for hull design. Then we can begin to expand the capabilities of this little language towards our engineering design purpose of implementing these rules in a nice (simple, maintainable, efficient) way.

**The simplest goal**  First we define the goal *eq* which attempts to make two variables equal by unifying them. This is the only type of goal which is not composed of other goals. The *eq* algorithm is as follows:

Source Code 8.7: *eq* algorithm to attempt unification and return unified (or nullified) state. Operator overloading of allows this method to be accessed with a call to Python’s `==` method.

```python
@staticmethod
def eq(a, b):
    def pursue(state):
        st = []
        if state is not None:
            new_state = state.unify(a, b)
            if new_state:
                return [new_state]
            else:
                return []
        return pursue
```

Using what we have shown so far, we can declare variables to have value within a given environment. We can use our *eq* goal to unify variables with each other, and with values. This goal will return a state where the value of a variable is set as desired, if in fact it is feasible to unify the value with the variable. Otherwise, this goal will return the empty list, indicating that the goal failed. An example of the Python code in action will demonstrate this simple goal:
In the code sample of listing 8.8, the basic eq goal is used to take two logic variables, \( x \) and \( y \), which evaluate to intervals in state \( st \), and unify them. The result is the intersection of the initial intervals. In this sample we’ve taken short cuts to show how a simple goal is dealt with by unification. Again, via operator overloading, the function is activated with the call to Python’s \( == \) method. Now that we have an elementary language with a procedure for asking that two variables be made equal, we can extend this to more complicated constraints.

On one hand, if following the development of miniKanren, it is now traditional to develop goals for the relational equivalents of logical and and logical or followed by a demonstration of something more often considered to be the domain of procedural programming, such as the append function, which can then be shown to “fill in the blanks” by running “backwards” [152]. On the other hand, here we are more immediately interested in relational mathematics for our ship hull rules. Accordingly, we will instead develop relational operators for basic mathematical operations. Along the way we will have to extend our relational capabilities from two terms, to three, and then to “n” terms. There are many ways to do this. In this work, we implement two functions (binop, run_binary_relation) which work together to handle outright simple mathematical ternary relational operators, add, subtract, multiply, divide, and then show how to extend these to relations with \( n \)-terms by using operator overloading and Python parsing to construct computational syntax trees.
Other approaches are certainly worth pursuing. Some relational constraints such as exponentiation or special functions could be the subject of further efforts. We will utilize only the simplest functions which enable our relational rules to describe ship hull geometry. Accordingly, the four primary arithmetic rules will be sufficient, when implemented as interval relational constraints. Next, for motivation purposes, we will describe those naval architecture rules in some detail. This will necessarily include some particulars of hull design, and automated hull design (form parameter) systems. These preliminaries will show what kinds of rules we need. Then we will show how to implement the primary terms.

**Bringing Intervals together with Constraint Programming**

**Rules Engine For Preliminary Ship Hull Design**

The merger of interval arithmetic and relational constraint rules systems gives a constraint logic programming system the ability to reason about a continuum of real values. This section explains how interval analysis (IA) is used with constraint logic programming to solve the problem of feasibility when going from continuous design specification to a set of form parameters upon which the curve design tools can act.

Design criteria are to be formulated according to a hull data model as input to the automated design system. This approach decouples design rules from logic which implements such rules. Further, the rules system is decoupled from the form parameter ship hull curve design framework itself.

A major advantage of the rules engine approach is that it is declarative. Users specify what the rules are, and the engine goes about enforcing them on the design space.

The rules approach makes it easier to express solutions to difficult problems and consequently apply these solutions. Further, a rules system makes design choices made by the program well-defined. By simply recording the rules as they are applied, one can store a well-defined explanation for how a design space was narrowed down. If a deterministic approach was followed for design space search, and not just design space inference, then all the choices that the machine made in designing the hull would be similarly easy to state.

Logic and data are separated. Basic data are stored as symbols with values. The logic is formulated as rules which are relations between the data. Actually applying that logic to the data is done in a decoupled fashion. In other words, all logic is applied by the same unification algorithm, developed separately from rules and data.

This is a fundamental difference from the standard object oriented approach, which would couple the
rules to the data more directly. In such a paradigm the rules and the procedures which implement the rules are exactly the same.

The result is all aspects of the problem can be maintained separately. By using a rule based approach for representing hull design knowledge, a repository of knowledge is created which can be modified independently of ship particulars or form parameters themselves.

Some Basic Naval Architecture Rules...

In many documents on ship hull design, from the introductory, as in Spoonberg’s [191], to the more involved and expert oriented, [192], or [193] several basic relations of naval architecture are given. These are elementary relations describing hull forms which are normally thought of as descriptors of a ship hull, more so than design constraints. But for purposes of telling our system what ranges of initial form parameters make sense together, we shall see that they are of the utmost importance.

The relations are to be detailed below:

- Rules :: Ship Design Requirements
- Particulars :: Ship Design States

Principal Parameters and Relations

Below are the basic parameters and relations used in the reasoning system. These relations are to be used in the computational rules graph to produce self consistent sets of form parameters. To avoid confusion with mathematical sets, this paper will refer to the “set” of valid form parameters used to initialize a form parameter ship hull design as instead a “tuple” of form parameters.

Principle Dimensions and Coefficients

Form Parameters

- $\nabla$: submerged volume
- $L_{OA}$: length overall
- $L_{WL}$: waterline length
L_{PP} : length between perpendiculares

B : beam

D : depth

T : draught

A_M : midship section area

LCB : location of the center of buoyancy of the ship

**Hull Coefficients**

C_B : block coefficient

C_P : prismatic coefficient

C_M : midship coefficient

**SAC Curve Parameters**

L_{FSAC} : length of the flat of water line curve.

**Water Line Curve Parameters**

CWL : water line curve

L_{FWL} : length of the flat of water line curve.

**Center Plane Keel Curve Parameters**

CPK : center plane keel curve

L_{FCPK} : length of the flat of the center plane keel curve.
Hull Form Relations

The rules and relations defined below are used in the constraint logic programming layer. They are implemented as a constraint graph.

The simplest rules are simply basic relationships from naval architecture. They express the logic of the ship hull. The system computes them as interval valued relations. These simple relationships, when combined into a network where interval valued data is allowed to propagate in any direction, will be enough to ensure that design space data can usually be narrowed in such a way as to produce feasible tuples of form parameters for the hull design system.

Note that in general there will be multiple relations for a given parameter. Each component of the relations below is interval valued. Every computation gives interval valued results.

All of the rules and design space choices below are experimental choices meant for modification. Adding, or taking away, a rule for the design space is trivial. However, the variable names themselves, and their existence in the design space, is “hard-coded” so that the curve solvers which actually generate the geometry will rely on the existence of ship design data with those names.

Rules and terms are from the following sources: [9, 52, 191, 194, 195]

Primary Hull Relations

Block Coefficient

\[
C_B = \frac{\nabla}{L_{WL} \times B \times D}
\]

Prismatic Coefficient

\[
C_P = \frac{\nabla}{L_{WL} \times B \times D \times C_M} = \frac{C_B}{C_M}
\]

Where \(C_M\) is the midship coefficient and \(C_B\) is the block coefficient (see below).

Midship Coefficient

\[
C_M = \frac{A_M}{B \times D}
\]
Waterplane Coefficient

\[ C_{WP} = \frac{A_{WP}}{L_{WL} \times B} \]

Centerplane Coefficient

\[ C_{CP} = \frac{A_{CP}}{L_{WL} \times D} \]

LCG constraint

The center of buoyancy must be within the length of the boat.

\[ LCB \leq L_{WL} \]

Or better, an LCB over \( L_{WL} \) ratio for finer control of the longitudinal center of displacement:

\[ LCB = C_{LCG} \times L_{WL} \]

Length to Beam Ratio

\[ C_{LB} = \frac{L_{WL}}{B} \]

The utility of ratio’s such as this becomes apparent as they say so much about the hull shape in such a small statement.

Displacement to Length Ratio

\[ DLR = \frac{\nabla}{(L_{WL})^3} \]

Flat of Curve Relations

Flat of the Water Line Curve

\[ L_{FWL} \leq L_{WL} \]
Flat of Centerplane Keel Profile Curve

\[ LF_{CPK} \leq L_{WL} \]

Flat of Sectional Area Curve

Length of the flat of SAC:

- \[ LF_{SAC} \leq \frac{C_P \nabla}{A_M} \]

Where \( ia \) functionally denotes an interval number by its Python class and heuristically setting: \( x_l = 0.8 \) and \( x_u = 1.0 \)

- \[ LF_{SAC} \leq L_{WL} \]

- \[ LF_{SAC} = L_{WL} \times ia (.06, .12) \]

The two rules immediately above highlight an interesting facet of the loose coupling inherent in constraint logic programming.

- These two constraints would make for a singular matrix if we were setting up a system of equations for traditional numerical analysis. (And the inequality became active)

- The constraint logic system could handle multiple equality constraints on the same variable with no complaints.

- Here in the user facing front end of the hull generation tool, we favor robustness and ease of use of speed, efficiency. Economy of description can be left up to the user.

Flat of Curve inter-relationships

- The length of flat of SAC should be less than the length of flat of center plane keel profile curve.

\[ LF_{SAC} \leq LF_{CPK} \]
• The length of flat of SAC should be less than the length of flat of water line curve.

\[ L_{FSAC} \leq L_{FWL} \]

• Keep the length of the flat of center-plane curve close to and at least as long as the flat of the sectional area curve, (SAC).

\[ L_{F_{CPK}} = L_{FSAC} \times ia (1., 1.1) \]

Miscellaneous Rules, Heuristically Chosen

Waterline Shape

New Rules to ensure good area constraints on DWL

• Obviously, the waterline area must be greater than the rectangular portion formed from the length of flat of the waterline curve multiplied with the beam.

\[ A_{WP} \geq B \times L_{WL} \]

• And the waterline area must be less than than the rectangular area formed from the length of the vessel at the waterline multiplied with the beam.

\[ A_{WP} \leq B \times L_{WL} \]

But this is taken care of by the coefficient of waterplane area.

• The waterline area should be great enough to handle the following components:
  - a full aft section (basically a box)
  - a box mid section,
  - and a triangular taper fwd to the bow

At the same time, there is no constraint specifying ahead of time exactly where the flat portion of the
waterline curve will be. Currently, this is all expressed as follows:

\[ A_{WP} \geq B \ast (LF_{WL} + ia (0.75, 0.75) \ast (L_{WL} - LF_{WL})) \]

The form above follows from the ambiguity of not knowing exactly how much of the waterline will be forward of the flat portion of the waterline curve, and how much will be aft of the flat portion of the waterline curve.

This rule is heuristic. Experiments have shown it do a good job of preventing there from being a need to have too little waterline area forward.

It should be noted that a better way to control the volume distribution fore and aft would be to explicitly denote the distribution of volume in three sections. Nevertheless, this way works pretty well.

Perhaps one could imagine updating the rules to explicitly solve for a 3 part waterline curve, 3 part sectional area curve, etc., and then make sure the current solvers can still work with the sum totals, plus resultant local constraints. This might be a good half step to get back to developing 3-part bare hull forms. (The present form parameter bare hull solver is a 10,000 line module which was developed over 6 months of painstaking trial and error. –Not a fun thing to modify, and really needed a major re-write - but that could take another 6 months.)

**Centerplane Keel Profile Shape**

- The center profile area should be great enough to handle the following components:
  - a roughly triangular aft section
  - a box mid section,
  - and a full but curved fwd section to the bow

At the same time, there is no constraint specifying ahead of time exactly where the flat portion of the center-profile curve will be. Currently, this is all expressed as follows:

\[ A_{CP} \geq D \ast (LF_{CPK} + ia (0.4, 0.4) \ast (L_{WL} - LF_{CPK})) \]

The inequality imposes a less restrictive lower bound on the area of the center-plane, as compared to
the waterline area, due to the fact that the center-profile-curve is curved both fore and aft.

**Initialization Relations**

Essentially the design space can be initialized by supplying reasonable values for those parameters not supplied by the user. For the examples to be shown in this thesis the following experimental inputs were used in formulating the design space.

\[
\begin{align*}
C_B & = ia (0.6, 0.75) \\
D & = ia (15., 23.) \\
B & = ia (25., 35.) \\
L_{WL} & = ia (110., 160.) \\
\nabla & = ia (1000., 35000.) \\
C_P & = ia (0.6, 0.68) \\
C_M & = ia (0.94, 0.99) \\
C_{WP} & = ia (0.92, 0.94) \\
C_{CP} & = ia (0.78, 0.87) \\
L_{CB} & = ia (55., 85.) \\
C_{LCG} & = ia (.45, .55) \\
C_{LB} & = ia (4.2, 5.2) \\
L_{F_{SAC}} & = ia (0.06, 0.12) \times L_{WL} \\
L_{F_{SAC}} & = ia (1.2, 0.8) \times L_{F_{SAC}} \\
L_{F_{CPK}} & = ia (1.0, 1.1) \times L_{F_{SAC}}
\end{align*}
\]

Table 8.1: Some Experimental Input Design Space Specification Choices

It’s worth noting that not all values need be specified. Table 8.1 above only includes inputs for some form parameters. The rest will be filled in as data and relationships dictate. Next we show relationships which are natural to use from the standpoint of basic ship hull form design.
Bulbous Bow Rules

Bulb-Bulb relations

Maximum Sectional Area Coefficient

This is the bulb sectional area at the forward perpendicular.

\[ C_{A_{mbulb}} = \frac{A_{mbulb}}{B_{bulb} \times D_{bulb}} \]

Waterplane Bulb Coefficient

\[ C_{A_{WPbulb}} = \frac{A_{WPbulb}}{L_{bulb} \times B_{bulb}} \]

Centerplane Bulb Coefficient

\[ C_{CPbulb} = \frac{A_{CPbulb}}{L_{bulb} \times D_{bulb}} \]

Bulb Block Coefficient

\[ C_{Bbulb} = \frac{\nabla_{Bulb}}{L_{bulb} \times B_{bulb} \times D_{bulb}} \]

Bulb Prismatic Coefficient

\[ C_{Pbulb} = \frac{\nabla_{Bulb}}{L_{bulb} \times A_{mbulb}} \]

Bulb-Bare Hull relations, Linear

- Breadth parameter:

\[ C_{bb} = \frac{B_{bulb}}{B} \]

- Length parameter:

\[ C_{lpr} = \frac{L_{bulb}}{L_{WL}} \]

- Depth parameter

\[ C_{zb} = \frac{D_{bulb}}{D} \]
Bulb-Bare Hull relations, Non-Linear

- Cross-section parameter:
  \[ C_{abt} = \frac{A_{Mbulb}}{A_M} \]

- Lateral parameter:
  \[ C_{abl} = \frac{A_{lateral}}{A_{CP}} \]

- Volumetric parameter:
  \[ C_{vpr} = \frac{\nabla B_{bulb}}{V} \]

Specific Bulbous Bow Design Space Choices Currently Imposed

- \[ C_{Bbulb} = ia (2.0, 15.0) \]
- \[ C_{A_{Mbulb}} = ia (0.5, 1.0) \]
- \[ C_{A_{WPbulb}} = ia (0.7, 0.97) \]
- \[ C_{lpr} = ia (0.03, 0.045) \]
- \[ C_{zb} = ia (0.2, 0.32) \]
- \[ C_{CPbulb} = ia (0.5, 1.0) \]

Table 8.2: Some Experimental Input Bulbous Bow Design Space Specification Choices

Other (Meta)Parametric Design Input Choices

- \[ \text{HyperParameters.fwd_fairness_location_ratio} = .36 \]
- \[ \text{HyperParameters.fwd_transition_location_ratio} = .75 \]
- \[ \text{HyperParameters.aft_transition_location_ratio} = .3 \]
- \[ \text{HyperParameters.aft_fairness_location_ratio} = .65 \]

So called “fairness curves” are the transverse curves which are located furthest aft, and furthest forward which are not actually forming a portion of the bow itself, i.e. the bulbous bow, or the stern itself, e.g. a stern propeller tube. The bow fairness curve will have great influence on the way in which shape flows from the primary hull into the bulbous bow, for instance, as well as primary influence on the volume distribution in the
upper bow. These effects, and the general control of bow shape by bow fairness curve, will be shown in the final results section of this paper. Transition curves are fore and aft transverse section curves which enforce the transition from the bow and stern to midship. Fractional “hyperparameter” inputs given here denote the locations of these curves as fractions of the total allowable translational range of position of these curves. That is, a value of “1.0” would move a curve to be coincident (or actually very near to coincident) with, in the case of a forward curve (either fwd_fairness or fwd_transition curve), the forward edge of the midship flat sections. On the other hand, a value of 1 would move an aft curve coincident with the aft-most stern transverse curve. A value of 0. for a “fwd” curve here would move that curve to be coincident at the keel with the bow center plane curve. So in these hyperparameter inputs, a larger fraction moves a curve aft.

Note that there are three more hyperparameter inputs used as well. Two of these are for bulbous bow design and integration with the bare hull, as described in section 11.2 and especially subsection 11.2. The remaining hyperparameter determines the final height of the center plane keel as a decrease from the height of the waterline. (This will naturally be reflected in the aft portion of the bare hull SAC curve, as it will become a form parameter, just like any constraint.)

The primary thing to note about these is that they could, and probably should be actual rules. As is they are simply input into the form parameter hull solver. These hyperparameters are as follows:

```plaintext
HyperParameters.bulb_volume_fraction  = 0.75
HyperParameters.bulb_length_fraction  = 0.3
HyperParameters.aft_drop             = 0.25
```

As is, they were built into the system as higher levels of detail became possible. As is, the aft drop parameter becomes a form parameter of the bare hull SAC, specifying the aft starting “height” of this curve, and also an important geometric position on the centerline transom of the hull. Further, the bulb volume fraction and bulb length fractions are used to produce subsidiary constraint logic rules for the bulbous bow rules consistency solver as described in subsection 11.2.

**Summary**

The above rules define the constraints on a design space. The design space is further defined by a starting set of interval inputs, one interval range for each design variable used in the rules. Note that a rule can also simply define the starting range of an interval. If a portion of that starting range declared in the rule is
already infeasible due to some previously defined constraint, then this partial conflict will manifest itself in the resulting design space - which will be appropriately contracted. The main point is that the starting interval ranges can be adapted by the designer (or rules creator) to the design problem under consideration, and that the designer does not have to worry about enforcing consistency across all intervals dimensions of the space, and inter-related rules. The system handles consistency automatically. If a completely infeasible space of rules and intervals is defined, the designer will be immediately notified since this will result in a design space with no size. An list containing one empty state would be returned.

Note also that neither the above rules (nor input intervals) do not limit what other relationships could be expressed using this system. The rules shown above were simply rules that were based on some relationships from basic hull design texts, [191] and a few ideas that occurred to the author after numerical experimentation. It was the intent of the software to make the input of new rules as painless as possible.

For concreteness, we now give an example of a rule and it’s entry into the system. Take the length to beam ratio, restated here again for convenience:

\[ C_{LB} = \frac{L_{WL}}{B} \]

And here it is in python, within the ship designer program and declarative constraint classes:

```python
# rule: Length/beam ratio
#
Clb = Clb == lwl/bwl
```

Including all the details we have:

```python
# instantiate the design variables if not already:
lwl = lp.PStates(name='lwl')
bwl = lp.PStates(name='bwl')
Clb = lp.PStates(name='Clb')
```
# program the Length/beam ratio rule here:
#
# Clb = Clb == lwl/bwl
#
#-----------------------------------------------
#
# compile this rule to the relational rules base:

hullrulesnet.set_rules(Clb)

The `set_rules` call wraps code to be explained below which will compile the rule into the system. With that compilation step, this rule becomes a list of relational function closures which await further computation.

In this case, since the rule is composed of so few components (only 3 variables), the list of function closures will be the same as the rule itself. That is, one function closure will implement this rule. We have already seen how binary equality relations are implemented. We will shortly give the implementation details to extend this to three terms, and basic arithmetic.

But when rules are composed of more than one ternary arithmetic relation, “connective terms” are needed to propagate information across even the rule itself, due to the fact that unification is binary, and relational arithmetic “always” works on a node which combines two operands into a result. The ternary relational rule structure we will implement below is not general, and one could build higher level structures to handle more and more operands, but in practice it is best to confine ourselves to rather simple elementary operators and generalize to higher numbers of terms in ways that do not require an ever higher tower of functions to be built by hand.

But for the moment we ignore implementation details, and concentrate on the action of the above rule on our design problem. When presented with a list of environments (design spaces as the engineer would say), this rule filters them, returning only the feasible space (of each environment) given the state of the environment before the rule, and after the rules effects have propagated throughout that state. Any environments without any feasible states left are eliminated entirely from the list of environments. Propagation is handled by maintaining a map of which variables are used in which functions, and thus when a certain function causes design space contraction, the program knows to call other functions associated with the variables which changed. The details of this will follow in the next few subsections.

Now that we have seen what kinds of hull design space relationships we desire to implement, next we
need to see how they can be used to narrow and split an interval design space (interval box) in order to find
the feasible domain effectively. To do that we must extend our relational interval language implementation.
So far we have shown how the basic “tiny relational logic language” was built. This program, as implemented
so far, works basically for equality unification of two term (binary) relations. Given the hull design relations
above, it is clear that more than two terms are needed for general rules. Now we will extend our relational
programming system with operators to facilitate ternary interval relational rules programming. The final steps,
after this, will be to extend the system to work for \( n \)-ary terms and also for automatic constraint propagation
across the entire rules set.

**Extending Unification to Ternary (Arithmetic) Relational Rules**

Arithmetic relations have been implemented by overloading the standard mathematical operators using
functions which reside within the `state` class. In relational form, these arithmetic constraints are ternary
operators. The initial goals of the implementation were to facilitate list based processing of states and to allow
data to flow in any direction through the arithmetic operators. “Any direction” basically means that from one
representation of a given ternary (in the relational sense) arithmetic operation, such as the addition of two
numbers equal to a third number, the program “is smart enough” to run that operation “forward and backward”
for all inputs and outputs. Note that forward-backward propagators where defined within the interval hull
consistency algorithm explanation in subsection ?? In this section we show how this implementation works.

To begin with, we show a simple ternary multiplication rule, here named “mullo” 8.9 after the naming
conventions of the miniKanren language [152] from which the system detailed herein took much inspiration.
This multiplication algorithm is implemented in the form of a closure so as to facilitate late binding of state.
13 Here once the multiplication rule is defined, it waits for state to be passed into it for the multiplication
computations to actually occur. Our implementation uses Python’s built in operator class to point to the
standard python operations, such as multiplication, which in turn pass control via operator overloading to the
constraint programming class and associated functions described above. There is still a mismatch between the
ternary multiplication relation and the underlying binary unification. In between the multiplication request

---

13Late binding, which means that the object being called, and the arguments it is called with, are “looked up” and “bound” to
each other at runtime, is an essential part of the mechanisms allowing this program to keep design state separate from design rules.
Essentially, variable and value are only combined or “bound” when specifically requested. Contrast this with standard python, where
once a variable is set, evaluating it in the interpreter always produces the set value. Python practices “early binding.” See the earlier
section detailing the design of the elementary logic programming system for details on the representation of states, variable, and
values in our relational rules system.
and the binary unification, is a ternary computation layer. First we will show the multiplication rule, and then describe the ternary layer that will allow connect it to the underlying unification and equality constraint.

Source Code 8.9: “mullo” a Relational Rule For Multiplication

```python
def mul(x, y, z):
    return mul(x, y, z)

@staticmethod
def mulo(x, y, z):
    def pursue(state):
        s = run_binary_relation(x, y, z, state, Goal.mul)
        return s
    return pursue
```

To implement such rules for the most basic arithmetic is fairly straightforward. Note that the mulo function is implemented as a closure, since it returns another function. This kind of programming practice allows mulo to remain ready to receive states. Upon a state being passed to the mulo function, it is processed, thereby filtering the state, and the filtered state (actually a copy of it) is returned. This copy also facilitates the “memory” of which relational logic programming is known for. Essentially keeps a record of its prior states.

Next we handle the mismatch between the binary unification process and the ternary multiplication rule. This requires the addition of a function, `binop`, to pre-process the arithmetic constraints, partially computing the relation so as to send binary data to the unification algorithm, and the use of the `run_binary_relation` to thoroughly propagate the data across all three components of the rule. (And especially to extend the computation to multiple states)

The generic ternary to binary function `binop`, shown at the top of listing 8.9 above, is borrowed from Mathew Rocklin’s implementation of mathematical constraints [153]. This `binop` is given in listing 8.10, below. The implementation of `run_binary_relation` is given in the appendix at listing 8.11

Note that in listing 8.10 above, the `goal` function, produced by `binop`, calls the `eq` function, which is part of the underlying elementary logic language defined earlier and the implementation of which was
def binop(op, revop=None):
    """ Transform ternary operator into goal """
    def goal(x, y, z):
        if not isinstance(x, Variable) and 
        not isinstance(y, Variable):
            return Goal.eq(op(x, y), z)
        if not isinstance(y, Variable) and 
        not isinstance(z, Variable) and revop:
            return Goal.eq(x, revop(z, y))
        if not isinstance(x, Variable) and 
        not isinstance(z, Variable) and revop:
            return Goal.eq(y, revop(z, x))
        else:
            return Goal.eq(None, None)  # (z, revop(x,y))
    goal.__name__ = op.__name__ + 'binop'
    return goal

given in listing. 8.7 As arguments, binop accepts a particular ternary logical rule, op, and in the case of multiplication mul is passed in, and its inverse, called revol. This will be a division function. The binop function then directs a partial computation of these rules, using a call of the op function, (or revop function ) in the definition and construction of the goal function, using the eq function to send a binary portion of the computation ultimately to the unification algorithm. The particular computation to be performed actually depends on the types of the variables which are used as input to the original mullo call. These become our x, y, and z variables here. Note that, via op(revop), the goal sends partially computed data into the eq goal. Note also that eq was described earlier in listing. 8.7 Thus binop will attempt to unify the constraint variables as used within the multiplication goal and return a newly unified (narrowed, i.e., reduced) state. Furthermore, binop is logically processing ternary data into forms that can be used by binary unification procedures.

Details of the binop function

- Binop takes the 3 logic variables which form a relation.

- Binop is another instance of “closure-like” operator. When instantiated in an actual arithmetic relation,
the procedures implementing the relation and it’s inverse are passed in as \( \text{op} \) and \( \text{revop} \), respectively.

- The approach encapsulated in \( \text{binop} \) is only natural for very basic operators. Even division must be flipped around to implement multiplication so as to use this successfully in all cases.
  
  - Consider division where \( y \) is unknown. In this case, a naive \( \text{revop} \) will not result in the correct answer.
  
  - In order to fix this situation, the division process must be “pre-flipped” so express multiplication, which can be inverted the same way, no matter which term needs to be solved for.
  
  - For more general operators, for example power, \( \text{binop} \) must be abandoned, and particular code to handle each case must be constructed with the operator itself. This fits with the operator overloading model used throughout this thesis however, and so perhaps a future version of this code will not use \( \text{binop} \). Nevertheless, for add, subtract, multiply and even divide, this function works quite admirably.

- To summarize, \( \text{binop} \) wraps the core arithmetic functionality in such a way as to naturally support two way computation even over ternary relations.

In summary, \( \text{Binop} \) is a sort of helper function which wraps the underlying relational arithmetic logic language goal, \( \text{eq} \) such that it is properly expressed no matter what kind of (allowable) inputs are passed into the ternary expression for the arithmetic operation. It is critical to this implementation that \( \text{revop} \) be the inverse operation to the actual operator, \( \text{op} \), since that actually enables the backward portion of the local algorithm to compute the correct inverse (essentially via hard coding, at the level of the overloaded function).

In practice, the simple set of operators:

\[ [+,-,\times,/>,<] \]

add, subtract, multiply, divide, greater than or equal to,
less than or equal to

has been found to be sufficient for the hull design rules used so far. Each has as its inverse one of the other operators in this set. Thus, like the interval design spaces themselves, the operators we use are closed.

A second function critical to the implementation of relational arithmetic is the \( \text{run_binary_relation} \) function which provides much simplified “run-like” (miniKanren run function) facilities on interval design
states. See the code for `run_binary_relation` in code reference 8.11 below. *The apparently redundant computation below is actually implementing full propagation of interval data across this ternary relation. The computation begins by passing fully walked values to the arithmetic relation (and thus to `binop`). Then `run_binary_relation` makes sure that one “non-walked” (i.e. un-bound) variable at a time is setup to pass through the `binop` function to arrive at unify ready to be unified with the full available information from the rest of the `binop` processed relation. For full propagation, this is carried out with each of the three terms in the relation. In future versions of this code, checks should be performed to stop the iterations when idempotent is detected within the automated contraction operations implied here.*

Source Code 8.11: Run Binary Relation: A function to extend binary relations to ternary

```python
def run_binary_relation(x, y, z, state, goal): #, option=None):
x1 = state.value_of(x)
y1 = state.value_of(y)
z1 = state.value_of(z)

states = goal(x1, y1, z1)(state)

ns = []
while None in states: states.remove(None)  # remove empty (None) states
  for s in states:
    ns += goal(x1, y1, z)(s)
    if len(ns)>0:
      states = copy.copy(ns)
      ns=[]
while None in states: states.remove(None)
  for s in states:
    ns += goal(x1, y, z1)(s)
    if len(ns)>0:
      states = copy.copy(ns)
      ns=[]
while None in states: states.remove(None)
  for s in states:
    ns += goal(x, y1, z1)(s)
    if len(ns)>0:
      states = copy.copy(ns)
      ns=[]
while None in states: states.remove(None)
  for s in states:
    ns += goal(x, y1, z1)(s)
    if len(ns)>0:
      states = copy.copy(ns)
      ns=[]

return states
```

To carry the explanation of the computations through, the `eq` function given in listing 8.7 ensures that the states are maintained as a consistent set of intervals. This code calls unify, given in reference 8.6 to actually
unify (via this case narrowing by interval identification) parameter states. This is the fundamental mechanism by which the constraint language is able to make logical connections and reductions on the design space.

Now we have the machinery for implementing our arithmetic relations. Below are two more arithmetic relations as example implementations.

**Addition over the interval states**

Addition, like all arithmetic constraints, is implemented with additional overloading structure on top of a miniKanren closure. (used the functools module’s partial capabilities). As shown below, the exposed method is a member of the `states` class.

Here is the code overloading part of addition in the states class:

**Source Code 8.12: Overloading Addition: relational on lists of states, member of the States class**

```python
def __add__(self, v):
    ""
    implements list based relational addition:
    usage:
    new_states = (old_state + (x,y,z))
    ""
    states = []
    for s in self.states:
        sn = Goal.addo(v[0],v[1],v[2])(s)
        for sni in sn:
            if sni is not None:
                states.append(sni)
    return States(states)
```

The code is as shown in order to allow for a certain syntax and especially to allow chaining together of state transformations. The code in listing 8.12 provides for syntax like the following:

**Source Code 8.13: Addition syntax**

```python
state2 = (state1 + (x,y,z) )
```

Or more generally, things like:

```python
state2 = ( (state1 + (x,y,10.)) + (x,y,z) )
```
As is, this function shown in code reference 8.12 will generate a closure like object that, when passed States, will narrow those states with the individual state implementations of add. The chaining and syntax shown in 8.13 exploits two things due to the fact that the closure takes a states object and produces the narrowed state once passed an initial state:

1. because the add function is a member of the states class, calling it as shown automatically ensures the state of interest is processed.

2. The result of the operation is another states object, this one guaranteed to have either the same state as before, or a narrowed version of it.

This allows for chaining of operations, and was useful only when composing constraint by hand programming. In the automated cases, neither the compiler nor the user will see code that looks like this. Nevertheless, all arithmetic functions used are overloaded in the same way.

Moving on to the rest of the addition implementation, note the code here depends on a function `addo`. This is part of the Goals class. Now the code for the states implementation of add is `addo`. This somewhat more functional (`Or at least less off the cuff`) code is implemented as follows:

Source Code 8.14: addo, member of the Rules class

```python
@staticmethod
def addo(x, y, z):
    """implements relational addition:
    usage:
    new_state = addo(v1,v2,v3)(old_state)
    ""
    def pursue(state):
        s = run_binary_relation(x, y, z, state, Goal.add)
        return s
    return pursue
```

This again acts like a closure. When passed a state the general arithmetic processing is handled in run_binary_relation. As shown here in 8.14, run_binary_relation actually takes a function Goal.add.

One way to tell python which addition we want to use at base is to use the operator module. Python will be smart enough to know that in addition to the overloading above, op.add here is actually going to point to the interval addition implemented elsewhere.

148
add = binop(op.add, op.sub)
add.__doc__ = """ x + y == z """

**Division over the interval states**

As shown below, the exposed (overloaded) division method is a member of the **states** class.

We start with the straightforward overloading part of addition in the **states** class:

```python
def __div__(self, v):
    """extended division on relational intervals this way doesn't work """
    states_ = []
    for s in self.states:
        sn = Goal.extended_divo(v[0], v[1], v[2])(s)
        for sni in sn:
            if sni is not None:
                states_.append(sni)
    return States(states_)
```

The code in listing 8.16 provides for syntax like the following:

```python
state2 = (state1 / (x,y,z))
```

Which can again be chained. The **extended_divo** function is the interesting one for `binop`. It is implemented as:

So there we see the trick which allows `binop` to work for this implementation of division. An added layer of indirection is used to reverse the division into a multiplication. Of course, multiplication has as its **revop** (reverse operation) division, but multiplication is “symmetric” under inversion. That is, I can solve the ternary multiplication relation by dividing by either of the multiplicands. Conversely, to try and invert the division operator would require either one multiplication or one multiplication and one division, requiring
mul = binop(op.mul, op.div)

@staticmethod
def extended_division(x, y, z):
    """ x - y == z """
    return mul(z, y, x)

@staticmethod
def extended_divo(x, y, z):
    def pursue(state):
        s = run_binary_relation(x, y, z, state, Goal.extended_division)
        return s
    return pursue

more specific processing than is available in binop.

**Important Note** Please note that subtraction requires a similar reversal in order to handle elegantly. That is, (a-b = c) is not symmetric under inversion! a = c+b b = -(c-a) Division and subtraction are antisymmetric in this sense, as far as the implementation of the reverse-operators (binop, revop) is concerned.

So now we see how simple relational arithmetic functions are constructed. Ternary relational rules can be formed from sequences of these directly.

But in general rules are more complicated. Next we shall use dummy variables to construct a more complicated rule by linking ternary operations together.

Now we have our arithmetic operators implemented in relational form on top of the underlying tiny relational logic language. But one arithmetic constraint in isolation is not going to be sufficient. We need all the constraints to work together. They will be tied together in two ways:

1. Constraint propagation across the entire design space of our rules.

2. Rules parsing for easy construction of $n$-ary terms.

Next we describe constraint propagation.
Efficient Constraint Propagation over the Graph of Relations

As seen with the closure property of the mulo algorithm 8.9, state must be passed through a rule in order to propagate interval narrowing data across a relation via the unification algorithm. Indeed, state must be passed through the rule before the rule will return anything at all! Unification reasons about a single rule at a time, given the state at hand. Therefore, some algorithm is needed to ensure that narrowing information can propagate across all the rules with some efficiency. The primary algorithm supporting propagation of updated state through multiple relations to various parameters is the arc consistency revision, or AC\textsubscript{revise} algorithm.

From the perspective of interval analysis, this algorithm extends consistency functions in an arc across any number of hull relations. One should think of this as the global extension of the local AC4 algorithm shown in figures 10 and 11. In the case of the work done so far, all consistency relations are of the hull consistency type, though there is no need for this peculiarity to remain so. Local box consistency operators, or any other narrowing operator, could be used as well. A global interval newton’s method might narrow the domain of one single constraint, and thereafter remain stuck, but there might yet be enough new local information to continue narrowing interval domains, and if so that could be found using AC\textsubscript{revise}.

AC\textsubscript{revise} is sufficient for our purposes, and would be needed in the case where we implemented an interval Newton contractor algorithm, to ensure progress in the case where the design space contained singularities. Therefore it makes sense to restrict our efforts here to AC\textsubscript{revise}. 
Source Code 8.19: Arc Consistency loop enabling non-local propagation of interval narrowing data

```python
def AC_revise(self, print_=False, maxit = 8):
    """Arc Consistency for the Hull Parameters
    """
    self.get_arc_sets()
    self.var_sets = set([])

    count=0
    while len(self.arc_sets)>0 and count<maxit:
        for arc in self.arc_sets:
            if print_ or self._verbose: print 'doing ', arc
            self.arc_sets = set([])
            self.get_arc_sets()
            self.var_sets = set([])
            count += 1
    return
```

The code in 8.19 is simply looping over arcs found in a set called `arc_sets`. This set is filled with relational functions where information is known to be ready for propagation. The key thing to notice here is that arc and variable sets are updated before and after the function loop. These notify the arc consistency function as to which variables have changed, and to which functions they are connected.

The system decides which relations to call as follows: The attribute `var_sets` is actually updated locally at the end of the computation of any relational constrain function only if one condition holds: If any relational computation narrows a form parameter, then that parameter is added to the `var_sets` set. Since it is a true set, variables can only ever appear in this list once.

Critically, each form parameter itself contains a reference list which contains every relation in which said form parameter appears.

Therefore, upon the computation of any relational constraint function, the form parameters are checked to see if they have narrowed. If so, that form parameter is added to the `var_sets` set.

The function `get_arc_sets` then traverses `var_sets`. Each variable in `var_sets` has a reference to every relation in which it appears. Thus `get_arc_sets` uses this information to build a list of every relational constraint in which new information is ready to be propagated by simply adding every referenced
constraint relation from all the form parameters in var_sets. The next iteration of arc consistency will use this list and update it again with a new set of variables which changed.

Before the loop over flagged constraint relations, but after the set of arc_sets has been built, the set of flagged form parameters is cleared. This way a new set of narrowed form parameters can be assembled during the constraint, or ‘arc,’ loop.

After the loop over the arc is complete, those arc are cleared from arc_sets and get_arc_sets is called once more to build a new propagation set from the freshly updated var_sets list.

Thus constraint propagation proceeds until either some maximum number of iterations occurs, or there are no more arc functions in arc_sets. Furthermore, we include a heuristic small parameter below which a particular interval is considered to be effectively narrowed. After all, the curve solver tolerance is typically around $1.e^{-6}$, which is a good bit wider than the interval system precision. Therefore we heuristically choose to cease narrowing our intervals at half of this width. In practice, a handful of iterations typically goes by before the narrowing subsides and the function loop exits. Occasionally, if a particularly burdensome choice were made for one of the form parameters, then the maximum iterations may be reached. In practice, this is quite rare. The far more likely issue in such cases is that the search will have issues finding a feasible design space at all. In such cases the culprit can usually be found amongst the form parameter design space choices setup by the user. This too, is rare, at least in our demonstration program, since an easy fix is to choose a starting space with wider intervals!

In the future this can be amended to include direct interval style checking to exclude infeasible subdomains. At present the check is implicit. If an infeasible state is reached during the narrowing of the form parameters, then that state is eliminated from the list of states by the unification algorithm. This method essentially relies on the unification procedure to discover non-unifiable states.

**Design Space Implementation**

With such a framework in place, the design space is initialized using a Python dictionary of name, value pairs. In fact, a list of dictionaries is kept, representing the possibility for generating disjoint design spaces during the constraint propagation phase.

Initially each variable name maps to itself, returning simply it’s own name when the database is queried for its value. When design specification is stated, this can immediately and straightforwardly be assigned into the ship design variables. Rules relate the state variables together locally, where relational constraint
propagation ensures that each rule narrows its associated intervals as much as possible, given the local information available in all the interval valued variables acting in the rule. Thus it is that relational rules bases are a natural setting for interval contractor programming.

Now that we have shown how to implement relational rules for interval arithmetic, and how to maintain consistency across a broad network of such rules, we need to show how to create such a network in such a way that users of the language don’t have to know everything about its implementation.
Chapter 9

An Internal Domain Specific Language for Constraint Propagation

The generation of a constraint rules system is a considerable amount of work within the programming system so far outlined. To make this system more tractable to program, maintain, and compute with for complex hull forms, we might like to streamline its use. After all, complex hull forms have many interdependent constraints, necessitating much boilerplate code in the system so far described. We can safely say that minimizing these encumbrances would be greatly helpful, and perhaps even essential, in moving forward to complex hull design. To see why this programming improvement quest is both a noteworthy and worthwhile endeavor, first we need to look at the requirements for adding a single rule to the system now. Then we may see that with only a few modifications to the system we have, we may produce a system which, instead of requiring the work which we are about to outline below, the system may automate away all of this leaving the user with only the responsibility for coming up with the rule itself. First let’s examine what we have to do in order to add a rule now:

To add a rule to a preexisting system of constraints usable in reducing a design space, a user must understand the following abstract issues and create procedures to handle them:

1. First of all, the new rule must be enclosed in a function that contains several pieces of boilerplate code. The function must:

   - Include handling for meta-variables (dummy variables) which connect constraints.
   
     - This applies to constraint functions which are larger than simple ternary relations between two arithmetic inputs and one output.
     - This means that when, e.g. two arithmetic relations are connected, ancillary variables, doing
nothing more than storing and propagating the partially computed information, must be created and managed within the states to correctly transfer the data between the physical design variables.

- Note this is a standard issue in constraint programming implementations.

- Include the necessary procedures to filter and return the design space using the new rule.

- Include the necessary logic to ensure that dummy variables used in the function do not conflict with those used elsewhere.

- One could attempt to keep all meta-variables local to particular constraint functions

- Without careful programming practices, this allows information to be lost if a certain variable is unified with a nullified meta-variable.

- Include some kind of flag to let other functions know when an update in this function might necessitate the calling of other functions, so that the data can be propagated.

2. Due to the last point above, it’s almost essential that the function reside in a class or within some other framework that ties all such rules functions together in such a way that when one function is called, the metadata can be used to call other function which use any of the same logical variables.

- This is required if any design intervals are narrowed since any design variable which has been narrowed could have a subsidiary effect in any other rule in which it is used.

The procedures implied by the issues above can be fairly opaque to programmers not familiar with the particulars of logic programming. Also, we’ve not yet shown how to make our rules language work for an arbitrary number of terms. The implementation of this will be described next.

In order to facilitate the extension of CLP beyond the feasibility of simple bare hulls, a simple “internal domain specific language,” or iDSL, was developed in Python in order to solve the problems outlined above. By using this language, the end user will avoid having to be wary of the issues outlined above and can instead focus more on the geometry and logic of the vessel hull form.

This will also have the side effect of naturally extending our capabilities from ternary relations to n-ary mathematical terms. The implementation will be given in the next subsection.
Implementation: Compiling Logical Rules

So far we have built the framework for a logic programming language. We could express all of our ship design rules in this language, but that would be cumbersome.

See the last section for details. We would like to automate these plumbing issues. In other words, relegate the onerous duties of managing dummy variables and boilerplate plumbing using a parser. An immediate issue with a new parser is that we wish to stay within the python language, and merely to add some new features to it. The solution is to build an internal domain specific language, or iDSL. If we keep the language internal to Python, there is no need to build a new parser or worry with “real language” issues at all.

Here is a scheme to use operator overloading to accomplish this:

- Use operator overloading to parse the rule statements as standard Python math.

- During operator overloading, create dummy variables as needed. Basically all operations between our logic variables should also return logic variables. Sometimes this will entail the creation of new variables. Sometimes it will entail using already created ones.

- Build up the computational graph as you would for reverse mode automatic differentiation. Each node in the graph (actually a tree) represents some mathematical operation with two inputs and one output.

- Walk the computational tree backwards starting at the terminating calculation node, assembling a ternary logical relation from the two inputs and single output at every node.

- Store these as the basic rule-building-blocks from which the complete rule is built.

- Store and update a map showing which rules use which functions so that the AC_revise algorithm will know which rules to call when a variable’s interval value(s) change.

Also note that it is advantageous to group elementary rules together when they form part of an overall rule as entered by the user. This helps computation to flow most efficiently locally through the rule as intended by the user.

The implementation consists of a class overloading addition, subtraction, multiplication, and division because at this time that is all that is used when defining the rules of our constraint network.¹

¹ This in no way means that we are limited to the basic four operations. In fact work in extending the supported constraints to
The overloaded class is constructed so that it will save its own computational graph (composed of computational syntax trees for each expression) as it is processed by Python. This is very similar to the way in which some authors implement automatic differentiation, as for example, Kearfott, et. al have done with (GlobSol [103]). When the syntax tree is walked from root to leaves in order to build up a derivative structure, the algorithm is called reverse mode automatic differentiation. This type of algorithm is of course famous in machine learning literature.

In the “old” way of generating logic rules, as described with the functional details above, we used the bedrock unification algorithm and related classes and methods in section 8.2 together with extensions to support arithmetic in section 8.3.4. With only these tools, the user writes a rule in bare form, and must then supply the needed boilerplate and connective code to ensure constraint propagation is successful across multiple rules. In contrast, the iDSL will ensure that each variable a the rule naturally becomes a node in the computational graph expressing the rule. Python 2 naturally ensures that the rule is properly composed, adding intermediate variables along the way. This is simply part of what a language does as part of its parsing phase. 3 The iDSL will take particular advantage of this fact, using an overloading class of arithmetic operations, and a class constructor (init) which constructs our own specialized syntax trees.

Here is how it works: The essential code is contained in the PStates class. Logic variables are declared of type PStates. The constructor, or _init_ method for the class, given in code listing 9.2, together with the operator overloading operations shown in code listings 9.3 and 9.4 ensure that compositions of PStates objects return PStates objects, as long as they compose with supported operations. As composition proceeds, each new PStates object assembles, in its constructor, the local information about how it was constructed. The critical data which is saved is as follows:

- **args**: the arguments which were composed together to create this new PStates object.
- **op**: the operation of composition.

---

2 The name Python is taken to mean the working Python interpreter, as opposed to the language specification, though that distinction may be somewhat arbitrary. The interpreter has built in support for expressing complicated mathematical forms, and everyone already knows how to use them. The geometry generation library described throughout this thesis is also in Python so it is natural to rely on Python again here.

3 Parsing is traditionally separated into two parts, lexical analysis (tokenization), in which the incoming character string is broken up into meaningful chunks, or tokens, and syntactic analysis, in which the tokens are assembled into an abstract syntax tree. We will be using operator overloading to essentially construct our own separate syntax trees.
Other data facilitates printing of the and other such local access. This primary data naturally allows the `PStates` object to become a node in a tree, as the argument list is a reference to other `PStates` objects or values. Note that values, such as intervals, are terminal nodes (also known as a leaf), and a `PStates` object can be a leaf, or a link to more `PStates` objects if it is the product of some composition. So, in this way composition of `PStates` objects build up a self organized syntax tree. Critically, this process will assemble a valid tree for any combination and number of arguments, so long as they are overloaded by methods of the class. Furthermore, Python will naturally assign intermediate variables to decompose n-ary compositions into partitions consisting always of two arguments and one resultant. Thus this overloaded parsing method is exactly what we need to extend our system to arbitrary mathematical forms, when suitably overloaded of course.\(^4\)

Now that we have assembled out syntax tree, here is how we compile it into a set of logic rules. The essential function is given in code listing 9.6, though note listing 9.5 provides support by predetermining which variables are to be used. Here is how compilation works: Let’s name a particular leaf in the computational syntax tree the “current leaf.” Each leaf in the tree is a node which contains two branches, representing the two intermediate variables which are composed to make up the current leaf. The backward pass algorithm descends the tree, at every node building a rules relation in the logic rules class based on the current node, and the two arguments (local leaves) from which it was composed using some particular operator, like “+” or “−” or “>,” etc, which is stored as well. By this chain of links each variable keeps a list of every logical relation that it is used to compose it, and exactly which operations were used, and in what order. Thus the code traverses the list, building logical function closures out of the ternary relations it finds in the tree. Furthermore, since by construction we know what variables are involved in which operations (and such variables are unique for all time), the code may simply add every new relation created to each variable’s particular, unique, list of relations in which it is involved. In this way if the interval data, which a particular variable represents, changes, then a program can easily find all the relations that can conceivably be effected in the next step of data propagation. The list of such relations is stored as part of the logical variable’s object data. In this manner, we have paved the way for “arc-consistency” and global constraint propagation.

\(^4\)Note also the contrast this with the “derivative tape” methods of reverse mode automatic differentiation.
# import our hull design code:
import relational_lsplines as rlspline

# our relational logic programming classes:
lp = rlspline.lp

# Make some design space variables:
lwl = lp.PStates(name='lwl')
bwl = lp.PStates(name='bwl')
draft = lp.PStates(name='draft')
vol = lp.PStates(name='vol')
disp = lp.PStates(name='disp')

Cb = lp.PStates(name='Cb')
Cp = lp.PStates(name='Cp')

# Make up a rule:
Cb = Cb == vol/(lwl*bwl*draft)

# Print the computational graph of the new rule:
print Cb
'Cb'=='
'v3'/
  'vol'

'v2'*
  'v1'*
    'lwl'
    'bwl'
'draft'

Source Code 9.1: “Pretty-Printing” the computational graph for a Cb relation, using the PStates __str__ method

As an example, the code shown in listing 9.1 uses the same computational graph structure just described to print back out the relation $Cb = Cb == vol/(lwl * bwl * draft)$. Parsing is handled “magically” via operator overloading on the logical variables class, of which each component of the rule above must be a member, as just described above. This example rule defines the block coefficient. It can now be compiled into a list of relational components. Every operation within the rule is composed of the combination of two terms to make a third term, “on the other side of the equation” so to speak. The user does not normally
think of a big expression broken up into little expressions this way, but we use Python’s parser plus operator overloading to do the same thing here, on our logical variable type.

So once the tree is built up via our overloading and PStates class composition operations, the “top node” is passed into another algorithm, which progresses down the tree from top to bottom nodes, and at every node enough information exists to build a relational rule out of the two leaves of the tree, which are the arguments, and the top node, which is the result (in procedural programming). Relational rules are composed in a continuation passing manner. This means that a when the program parses the node tree, it finds an operation combining two leaves into the local top node, that operation is really represented by a function which returns a function. This final function is left unevaluated until it is “fed” a States object. Instead, for now, it is appended to a list of functions defining a complete rule, and the rule structure is appended to a list of all the rules. 5 As might be expected, keeping the rule structure in which relations came “prepackaged” 6 was found to be helpful in ensuring more efficient constraint propagation through the network. In keeping with this idea, rules are compiled into a list. It might be worth noting that this incomplete functional evaluation, and placing of rules into a data structure for future evaluation, meets the characteristics of a simple compiler. 7 This compiler compiles function continuations which will accept a state as input and return a new filtered state as output. All connectivity is handled by the nodal variables themselves. When a variable has its interval value narrowed, an arc consistency function can ask it for its list of functions in which it is used. The arc consistency function calls those functions in turn, filtering States information through them and updating other variables, which again have their own lists of associated rules.

One crucial technical detail is that our rule is always processed into operators that are either binary assignment, or ternary computation. When dealing with an expression of greater than two operands, the parser naturally breaks the expression by the operator precedence which exists “naturally” in Python. Thus thanks to operator overloading, Python handles precedence for us. Next, if the terms represent some part of the full expression, the parser will naturally assign them to some intermediary value. Our algorithm gives this term a unique name as if it were a logic variable just like Cb or lwL, but instead of representing a form parameter we care about, it is just a connective term allowing the numerical data from ternary expressions to

---

5 Actually these structures were developed with lists of dictionaries.
6 I.e. if the user inputs a rule which is composed of compound statements, this becomes a structure which ensures the “atomized” rule components are all evaluated sequentially
7 Of course, this is not a very smart or powerful compiler. It does not even recognize all forms of mathematical expression, requiring all rules be written in the form of an assignment into one variable on the left hand side BNR Prolog, for instance, allows for less restrictive expression formatting, potentially requiring less term-rewriting by end users. [196]
propagate up the computational graph into yet more complex statements. In this way we never need to handle operations of greater than 3 elements and one operator, and we never have to write boiler plate connective code.

Now adding a constraint can be as simple as expressing the mathematical relationship in a simple, mathematical form, and passing the “top node” (variable) to the rules compilation engine. The engine processes the rule into a list of relational functions, adds any necessary intermediate nodes to the environment, and adds the relations themselves to the list of associated relations maintained by each rule. All of this compilation process is hidden from the user, who may simply ask the constraint network to update itself when new rules are added, or when variable values are set. Of course that too may be automated. The primary task of the user, in this system, is to discern what the relationships between the design variables are.

This simplification in programming is greatly needed when moving to accommodate ever more detailed constraints into our hull form generation system. This will be of great benefit when incorporating the bulbous bow, but more importantly, as with automatic differentiation, this style of programming, where users can compose problems on the fly, leads to a more flexible solver that can tailor itself to the particular problem at hand. This flexibility of program in the end gives rise to a more robust optimization system.

A resulting example rules net has been generated from a subset of the naval architecture rules given in section 8.3.2. Figure 9.1 shows the graph. This graph shows a basic set of hull form relations. A box represents a rule that has been added to the system. The boxes are also form parameters, but they are defined, at least in part, by the rule shown here. They can also be given starting interval values and set equal to anything else desired, though if it is incompatible with other rules or intervals, a null result will be obtained. Thus any form parameter can be assigned an interval, and it may also be assigned as many rules as the designer would like. During a consistency computation data would propagate through a rule in a manner like that of the local arc consistency algorithm shown in code listing 8.19 and introduced as an interval technique in figures 8.2 and 8.4. Note the apparent simplicity of the user experience compared with procedural consistency implementation such as that in [173], or the underlying implementation given here. By using unification and declarative programming, we simply ask the language to “unify” the terms found in each relation, no matter what they are. Arc consistency is computed by looping over the terms which have recently changed, and calling the functions in which they are used together with our current design environment.
Figure 9.1: Example Computational Graph Using Naval Architecture Rules. Ellipses represent design variables which have not yet been used by the designer as explicit rules. Boxes represent those form parameters which have. (Any mathematically defined term is also a rule. Computationally, if only for graphing purposes, we distinguish here between those that have appeared on the left hand side of our relational equality “double equals sign” and those that have not yet done so.) Lines represent paths of mutual dependence between form parameters. Self intersecting lines represent relations that also define design variables.
Source Code 9.2: `Pstates` Class: Code for Compiling Python Rules into little logic language rules

```python
class PStates(object):
    
    """
    Function
    """
    
    class which uses Python itself to construct
    computational graphs (mathematical syntax trees)
    of expressions input by user.
    It is like a first stage of a reverse mode AD tool,
    but for other things than differentiation.
    
    # The purpose of the Rules Graph Processor is
    # to automate this kind of thing (AC_revise), and to
    # make it more efficient, together with the compiler,
    # which adds links from mkVariable
    # to its own func-list of rules (where it gets used)
    """
    count = 0
    def __init__(self,
        args=[],
        op=None,
        opstr=None,
        name=None,
        value=None,
        verbose=False):
        if op is None:
            op=PStates.__init__
            opstr = 'init'
        self.verbose = verbose
        self.value = value
        self.op = op
        self.opstr = opstr
        self.args = args
        self.nodelist = []
        #list of this vars Big Rules {}
        if name is None:
            PStates.count += 1
            self.name = 'v'+str(PStates.count)
            self.discount = True
        else:
            self.name = name
            self.discount = False
```

The code in listing 9.2 shows how we implement the basic rule variable class, `Pstates`. Instances of
this class will be nodes in the computation tree. The computation tree will build itself when these variables
are used in the manner of standard Python floats.

Source Code 9.3: Pstates Class: Code for Compiling Python Rules into little logic language rules

```python
def __str__(self, level=0):
    """Fancy printer...
    walks the computational graph, printing nodes and their
    associated ops
    """
    ret = " " * level + repr(self.name) + ""+self.op.__doc__+"\n"
    for el in self.args:
        if isinstance(el, Number):
            dc = PStates(value=el)
            dc.name = str(el)
            #ret += " " * level + str(el)
            ret += dc.__str__(level+1)
        else:
            ret += el.__str__(level+1)
    return ret

def __repr__(self):
    return 'PStates({})'.format(self.name)
def equal(self, v):
    """==""
    return PStates(op=PStates.__eq__,
                   opstr = '==',
                   args = [self,v],
                   name = self.name)#,
    #value = v)

def __eq__(self, v):
    """==""
    #print 'v',v
    return PStates(op=PStates.__eq__,
                   opstr = '==',
                   args = [v],
                   name = self.name)
```

The code in listings 9.3 and 9.4 primarily show how we implement the arithmetic methods of this class.
Each time a Pstates object is used in an arithmetic operation, no arithmetic resultant is computed. Instead,
a record of the objects being combined in the arithmetic operation becomes encapsulated as knowledge within
the tree formed by the Pstates objects so combined.

Operator overloading handles the order of operations of the mathematical statements. The resulting
Pstates object is returned. This node has a link back to the inputs which formed it capturing the operation
of combination, and the input components, whether Pstates objects, or Python floating-point numbers.

Finally, the __str__ function is overloaded to pretty print the computational graph so constructed.
def __le__(self, v):
    """<="""
    return PStates(op=PStates.__le__,
                   opstr = '<=',
                   args = [v],
                   name = self.name)

def __ge__(self, v):
    """>="""
    return PStates(op=PStates.__ge__,
                   opstr = '>=',
                   args = [v],
                   name = self.name)

def __add__(self, v):
    """+"""
    return PStates(op=PStates.__add__,
                   opstr = '+',
                   args = [self,v])

def __sub__(self, v):
    """-_"""
    return PStates(op=PStates.__sub__,
                   opstr = '-',
                   args = [self,v])

def __mul__(self, v):
    """*"""
    return PStates(op=PStates.__mul__,
                   opstr = '*',
                   args = [self,v])

def __div__(self, v):
    """/"""
    return PStates(op=PStates.__div__,
                   opstr = '/',
                   args = [self,v])
def construct(self, treenode, st, existing=None):
    """Takes a computation tree built by overloading
    and returns 2 things:
    (1) vars: a set of sqKanren.Variable class (logic) variables
    (2) environment: sqKanren.States which represent
        the current environment.
    inputs
    --------------
    treenode : PStates object, possibly after building some
        computational graph by using it in an expression
        with other PStates objects or interval constants
    st : States object, an environment in which the
        results of assignments and other computations
        filters through.
    existing : if one already has started composing
        a dictionary of vars_ in use, then pass it in here.
        The map should look like: {PStates.name : Variable}
        where Variable is the Variable class from sqKanren.py
    returns
    -------------
    st : States object, updated with data constructed here
    existing : map of PStates object names to Variables.
    ""
    if existing is None:
        existing = {}
    if isinstance(treenode, Number):
        cur_el = treenode
        existing[treenode.name] = cur_el
    elif not treenode.name in existing:
        cur_el = st.bind(treenode.name)
        existing[treenode.name] = cur_el
    if hasattr(treenode, 'args'):
        for el in treenode.args:
            self.construct(el, st, existing)
    return st, existing
def compile(self, treenode, vars_, funclist = None, i=0, st=None):
    """construct the space with
    logic vars (use construct, above)
    before calling Goal.eval to
    construct the rules themselves.
    ""

    if funclist is not None:
        i=len(funclist)
    else:
        funclist = {}
        i=len(funclist)
    if len(treenode.args)>0:
        if len(treenode.args)==1:
            op = treenode.opstr
            operands = (vars_[treenode.args[0].name],)
            if self.verbose:
                print treenode.args[0].name
                print treenode
                print 'ops (1):-',op, operands
        if len(treenode.args)==2:
            op = treenode.opstr
            operands = (vars_[treenode.args[0].name],
                        vars_[treenode.args[1].name])
            if self.verbose:
                print 'ops (2):-',op, operands
            result = vars_[treenode.name]
            if self.verbose:
                print 'ops (F):=>','(',op, operands, result,')'
            #transform the rule into a function:
            func = Goal.eval(operands, op, result)
            assert (func is not None)
            funclist[i] = func #put the function on the list
    for el in treenode.args:
        if not isinstance(el, Number):
            funclist = self.compile(el, vars_, funclist, i+1)
    for var in vars_:
        mk_var = vars_[var]
        if isinstance(mk_var, Variable):
            #mk_var.flist.append(funclist) #code list onto mkvar
            for func in funclist:
                fi = funclist[func]
                mk_var.arcs.append(fi)
    return funclist
Compilers, as defined by Norvig in [150], translate some set of rules statements in one language into a program in another language. For our purposes, both source and target languages are internal to Python. Our compilation process can be visualized as in the chart above in figure 9.2. Note that the resulting code waits for an environment in order to be evaluated. This is a form of lazy evaluation.

To summarize what was said above, the rules compiler given in listing 9.6 accepts a computational graph (tree) assembled naturally by Python parsing together with the operator overloading of the PStates class. The construct function above in code listing 9.5 facilitates compilation by pre-computing the needed variables and environment. The compile function returns a rules dictionary each rule dictionary containing all the “atomic” relational rules used in composing an overall rule. These “atomic” expression were defined in the Goal class. Each atomistic goal is a function closure as defined earlier. Each goal is capable of filtering states, processes and returning new states with the updated consistency data. That is how a goal enforces its meaning on the environment. Note that the data structures returned by compile define the connections between atomistic rules as well. These are encapsulated in lists carried by the logic variables. These lists also specify which rules a particular logic variable is involved in. Furthermore, these connections between rule and logic variable allow the AC revise function to propagate constraint information across the entire graph using the mechanisms described earlier and implemented in 8.19.

This concludes the development of the relational language for ship hull design. Using this language, the program is able to filter out inconsistent form parameter design combinations by interval contraction and constraint propagation. These concepts and this implementation thereof forms a rudimentary global consistency framework for form parameter design. Nevertheless, a truly robust system should also be prepared
to handle local consistency issues. These might crop up from time to time if, for some reason, an onerous set of differential, positional, and integral constraints is imposed on a set of inter-related curves, in addition to the integral relations we have specifically tried to account for in the section above. Accordingly, we next extend form parameter design using the truncated hierarchical B-splines in order to allow for some measure of adaptivity of resolution at curve solve, and surface representation, time. In the end, and to allow for the display of the final geometry by the computer graphics software Rhino [197], we can simply project the THB-spline representation to the finest B-spline level, where all detail will be captured. This will also serve as a possible branching off point for more research into multi-resolution methods for automated ship hull design and representation.
Chapter 10

Truncated Hierarchical B-splines

Introduction to Multiresolution Modeling

Computer aided design / computer aided engineering (CAD/CAE) systems are becoming more integrated with analysis software via isogeometric analysis as well as extended to incorporate adaptive and flexible methods of model representation. Future CAD/CAE systems should provide for exact geometric design as well as facilitate direct physics modeling and analysis, and yet be flexible enough to facilitate the composing of complex models as well. Ongoing research and development in CAD/CAE has focused on schemes which maintain the properties desired for analysis, while at the same time providing the flexibility needed for complex and locally detailed model generation. In addition, multiresolution modeling is an asset to any geometry generation program. This chapter makes the argument that multiresolution modeling is complementary to other methods of robust design shown in this paper, ensuring more robust design by, in this case, providing for variable degrees of freedom where extra shape control might be needed. This extends form parameter design to model complex shapes with fewer basis functions. A curve solution is, in general, a non-linear process. We Make the case for attacking this problem using the THB-splines to supply extra assurance that the desired geometry can always be met.

Traditional CAD software systems provide a graphical user interface that allows design engineers to shape and otherwise manipulate model geometry via a set of operations and the standard representation has been the tensor product B-spline curve and surface in many cases.

One of the nicest aspects of the B-spline representation for geometry is the complex hull property, which enables intuitive manipulation of the B-spline curve via a parsimonious set of control vertices. A downside to the traditional B-spline representation is apparent as soon as one moves from a curve to a surface. Here, the
tensor product formulation prohibits local refinement of the control vertex mesh. This uncontrolled spreading of the detail coefficients away from the areas where they are actually desired has the detrimental effect of increasing the likelihood of spurious oscillations occurring during an algorithmic geometry fitting process. There is a balance to be struck between having enough control points to properly fit a set of data, and having a parsimonious set of points to keep things smooth and “fair.”

When moving to a constrained optimization setting as described in the other chapters of this text, there are further balancing acts to perform with regard to having enough degrees of freedom to meet all constraints. In a completely automated hull form generation setting, the program is additionally responsible for coming up with those constraints in the first place, and deciding where to put various geometry defining curves in relation to one another and deciding exactly how many transverse curves there should be in order to provide enough definition for the desired geometry.

Transverse curve placement, for example, can make the difference between a viable hull or a poor one when lofting a hull form through a set of transverse curves via the automated methodologies outlined beginning with the work of Harries and Nowacki [9,52] and continuing to be used for the fairness optimized transverse control vertex interpolating lofting procedures shown in this dissertation. In the automated setting, the program must choose stations at which to generate transverse curves, and then longitudinal curves must be generated that exactly interpolate the transverse curves so as to finally generate a surface which meets a set of form parameters exactly. In practice, an automated tool computing these procedures may start with a perfectly compatible set of form parameters, and generate a fair set of transverse curves from that, but the placement of those curves at certain stations along the hull length might lead to a system that is difficult to solve for in a smooth and fair manner. Here one could imagine a design loop which repositions the transverse curves in order to search for stations which allow the generation of nice longitudinal curves, but even then, there may be trade-offs as one set of stations leads to “better longitudinals here, worse longitudinals there” and another set of stations might achieve the reverse. Furthermore, this would be an expensive search as each new station tested would require at least one transverse curve to be freshly generated, and an entirely new set of longitudinal curves to be lofted. If instead the geometry representation has some local adaptive capability this would clearly benefit the hull form generation tool. Local adaptivity could ensure that all longitudinal curves are fair and exactly interpolating, with no search for the best transverse stations needed. This is what we intend to show here.

Furthermore, there is often a need design features into a hull which are small compared with the rest
of the hull shape, such as bow thruster ports or appendages, and a need to quickly substitute macro scale features, like a bulbous bow, where an optimization algorithm would benefit from the capability to modify this geometry independently, while at the same time there needs to be a fair interface between the bare hull and the geometry. Here a multi-level, or topologically adaptable modeling procedure would be quite useful.

In terms purely of automated complex geometry generation, multi-resolution exact representations of complex geometries are needed to both represent the geometry and manage and adapt the complexity associated with its definition while maintaining an ability to modify it algorithmically or by hand. Yet in general, an automated design program does not know in advance the best representation for a geometry specified by constraints. Thus the rigidity of traditional B-spline based frameworks work against an automated hull form design tool in two ways.

1. Complex geometry: Adding a control point in one location on the surface necessitates the addition of an entire row and an entire column of knots, and their control vertices, into the control vertex mesh. This makes hull features and geometric details more expensive to add to a model.

2. Sensitivity to representation: For example, placement of transverse curves at stations along the longitudinal length of the hull to be generated can greatly affect the viability of the eventual shape. For instance, a number of control points will be required to enforce various constraints along the hull, and yet too many control points will be difficult to fair. In general, more flexible curve and surface representations will provide for more desirable geometric outcomes.

This chapter makes the argument that adaptive multi-resolution schemes can assist an automated hull generation tool with both complex hull form design itself, and with ensuring that any mathematically feasible set of constraints, and associated intermediate curves, can actually be met/interpolated in a smooth and fair manner. This will benefit the form parameter ship hull design process by avoiding, for example, the need for bespoke placement of transverse curves for bare hull form definition via lofting with longitudinal space curves.

The remainder of this section is structured as follows. In subsection 10.2 we briefly discuss aspects of multiresolution and topologically agnostic geometric design and settle on the THB-spline representation. Then we recall the basics of THB-spline formulation and evaluation. In subsection 10.3, we describe multilevel editing. Subsection 10.4 is concerned with using the THB-splines in an automated and adaptive curve design setting to ensure robust construction of ship hull form geometry. Subsection 10.5 concludes
with further steps required to integrate the adaptive THB-spline optimization procedures into a robust surface construction and refinement process.

**Choosing a Representation And THB-spline Theory and Implementation**

**Beyond B-splines - Selecting a Representation**

First we state the reasons which will drive our choice of multiresolution geometry representation.

1. NURBS are an industry standard. We would like to remain compatible with them. Specifically, we require that our representation be everywhere compatible with B-splines as they are used elsewhere in this work for ship hull form representation and optimization.

2. Issues with B-splines and NURBS:

   - A single NURBS patch is either a topological disk, tube, or a torus.
   - Multiple patches are needed to model complex geometry requiring patch management, gluing conditions, and the like. This makes simple processes like deforming the shape by moving control vertices, complex, and also complicates constraint satisfaction.

3. Computational Properties Desired:

   - Efficient Evaluation (of the geometry and it’s properties)
   - Compact Support
   - Affine invariance
   - Partition of Unity/Analysis Suitability
   - Smoothness.
   - Ease of evaluation of derivatives and integrals of the geometry. (arc length, tangency, curvature, area, volume, centroids)

As noted throughout the literature, the initial development of isogeometric analysis was based on NURBS, which is based on a global tensor produce structure and thus does not support local refinement. Various spline techniques have been developed to address this shortcoming. They include hierarchical B-splines [34–37], T-splines [38, 39], and LR splines [40, 41].
From there, two essential branches can be seen forming in the spline literature. On one hand, subsequent refinements of the hierarchical idea have been ongoing, such as polynomial splines over hierarchical T-meshes [42], and truncated hierarchical B-splines [43, 44], spline forests, and truncated Catmul Clark subdivision [45] On the other hand, T-splines have been further developed as well. See analysis suitable T-splines [46, 47], modified T-splines [48] and truncated T-splines [49]

A hybrid scheme can be seen with hierarchical T-splines [50].

Subdivision surfaces are perhaps the chief competitor to splines. In the computer graphics industry, they have become the standard due to their natural topological flexibility. (In fact they support arbitrary topological meshes, as long as some continuity and integrability constraints can be relaxed) Catmul-Clark surfaces, Doo-Sabin, and Loop subdivision round out the most well-known variants. Stam, [27] showed how to evaluate the limit of a subdivision scheme without excessive recursive refinement. Later, methods of computation of various properties of subdivision surfaces have been shown by [198] and other researchers. Nevertheless, from an engineering standpoint, subdivision surfaces loose their nice computational properties, and the evaluation of geometric constraints breaks down, for some constraints, at exactly those areas of the mesh which topologically supersede the spline based methods. Furthermore, with the previously cited recent developments of spline based technology, there is less worry about meeting the demands for evaluating splines on domains with T-junctions, etc., by using advanced splines. Therefore, we choose to stay with spline technology when extending form parameter design to complex and multi-resolution modeling.

That leaves a choice primarily between T-spline and hierarchical technologies. Meshing refinement and isogeometric performance have been studied. See for example [199].

The choice came down to elegance of implementation, where THB-splines have the edge, as T-spline implementations with analysis suitability are somewhat cumbersome [46], parsimony of refinement, where T-splines seem to have a slight advantage [199], and a view towards future isogeometric applications, in that T-spline refinement, with its cumbersome system of rules for refinement, is especially burdensome after moving to 3D [46, 200, 201], or even when changing the degree of the spline [201]. Studies have examined the strengths and weaknesses of the various hierarchical schemes as well [41]. Also, there are well known commercial CAD package devoted to T-splines associated with the software packages Rhino and also with AutoDesk. In the end, it is the programmers choice and THB-splines are much more elegantly built than T-splines. With an eye towards managing complexity in general, and nod towards the luxuries of liking one’s tools, and the generality of restriction and prolongation projection methods, THB-splines were selected.
**THB-splines**

Truncated hierarchical splines (THB-splines) were introduced by Giannelli, Juttler, et al. in [43]. The THB-splines are descended from the original hierarchical B-spline work of Forsey and Bartles [34], who first described a multi-level B-spline system for hierarchical surface definition in which the fine levels overlapped the coarse levels, and then Rainer Kraft, [35], furthered the development of hierarchical B-splines by developing a global selection mechanism to select linearly independent basis functions from different hierarchical levels. The truncation mechanism, [43] adds the satisfaction of partition of unity to the list of properties that can be satisfied in the most well-developed hierarchical B-spline constructions. This last property was fairly important since affine invariance is lost without it because without partition of unity, the curve is no longer confined to the convex hull [51].

The basic form of a THB-splines evaluator is seen by examining the process of evaluation at one level of the hierarchy:

First we discuss evaluation for curves. There are actually any number of ways to construct THB-spline evaluation for curves, due mostly to the fact that a B-spline curve can already be locally refined. We shall concentrate on the evaluator that most closely parallels that of surfaces, which are more exacting in their requirements for local refinement.

To be as efficient as possible, from an evaluation point of view, evaluation should focus on vertices as opposed to basis functions, because the former are invariant with respect to parametric location, whereas the latter are functions of it. This is particularly important if you are going to evaluate a property such as the area of a THB-spline curve, you would be better off projecting vertices to the fine level and evaluating the area there, as opposed to projecting basis functions which will vary in the parameter space, or composing the basis functions on each individual level, and reconstructing the global area integral from the individual hierarchical components.

Of course when one refines a basis function via knot insertion, one also inserts a control point for every new basis function, so the THB-spline literature can be read as describing control point subdivision just as well as basis function insertion/subdivision. Following the literature, we take this view implicitly, and describe basis function refinement and truncation. The main trick when reading the literature is to operate on the control vertices, but talk about basis functions. The geometry is the same either way, but computationally and from a code development standpoint, this choice makes all the difference. Note that the viability of
refining directly on control vertices as opposed to basis functions is supported in the literature. See [202], for example. Also, by projecting all the vertices, instead of the active set, we enable the evaluator to be generalized to integral constraints.

That was a bit of a digression. Let’s make things concrete with some preliminary definitions, then the evaluation procedure, and then a Python implementation of a simple THB-spline curve evaluator.

Define a sequence of (spatially) nested axis aligned parametric domains (Note that this is also the parametric domain of the knot vector):\[
\Omega_0 \supseteq \Omega_1 \cdots \supseteq \Omega_N \tag{10.1}
\]

With 10.1, we are saying that higher parametric domains are always nested within lower level domains, but that the domain of a higher level can encompass the entirety of the level just below it, and so on.

Additionally, let there be a finite sequence of nested tensor-product spline spaces.\[
V_0 \subset V_1 \cdots \subset V_N \tag{10.2}
\]

A spline space at a higher level is “finer” and contains more basis functions that any particular space below it. We are saying in equation 10.2 that the basis space of a lower level is a subset of the basis space of a higher level. This makes intuitive sense because the higher space can always match the details of the lower space, but never the other way around.

Since we are dealing with spline spaces, the overall parameter domain is a rectangular domain. Without loss of generality we can parameterize the outermost spline space from zero to one inclusive.\[
\Omega^0 = [0, 1]^D \tag{10.3}
\]

In equation 10.3 above, $D$ is the dimension of the spline space. For curves, we have $D = 1$ and surfaces $D = 2$.

Furthermore let
be the normalized tensor-product B-spline basis of the space $V^\ell$

We denote the column vector of basis functions as

$$b^\ell (x) = \left( \beta^\ell_i (x) \right)_{i \in I^\ell}$$

We will additionally have a coefficient (control vertex) matrix $C^\ell$ and so the final evaluation of the spline is the standard dot product method:

$$s(x) = \sum_{i \in I^\ell} \beta^\ell_i (x) c^\ell_i = b^\ell (x)^T C^\ell$$ (10.4)

Here everything is entirely standard to B-splines, except that projection of the coefficients from level to level, and truncation of some of them, will be handled in the THB-spline manner, that is by using truncated refinement via projection, to be detailed next.

**THB-spline projection for evaluation**

The THB-splines are recursively constructed, so it is sufficient to study the construction of level $\ell + 1$ from level $\ell$. The construction mainly consists of three steps:

1. Determination of the level $\ell + 1$ domain $\Omega^{\ell+1}$
2. Selection of the level $\ell$ and level $\ell + 1$ basis functions.
3. Truncation of unnecessary level $\ell$ basis functions.

Before showing the actual algorithm to project THB-splines control vertices (scaling coefficients) and thus enable evaluation of the THB-splines, we first show truncation by itself, to highlights its appearance in the final algorithm.

Truncation discards basis with active children from the refinement. Giannelli, et al. [51] give the following construction for THB-spline truncation:

$$\text{trunc}^{\ell+1} s(x) = b^{\ell+1}(x)^T \left( I^{\ell+1} - X^{\ell+1} \right) R^{\ell+1} C^\ell$$ (10.5)
Where $I^\ell$ is the set of active basis functions on at spline space level $\ell$, and

$$X^\ell = diag \left( x^\ell_i \right)_{i \in \mathbb{N}},$$

is a diagonal matrix where

$$x^\ell_i = \begin{cases} 
1, & \text{if supp} \beta^\ell_i \subseteq \Omega^\ell \land \text{supp} \beta^\ell_i \not\subseteq \Omega^{\ell+1} \\
0, & \text{otherwise}
\end{cases}$$

and finally $R^{\ell+1}$ is a refinement matrix which projects the level $\ell$ basis/or coefficients to the level $\ell + 1$ level.

Note that one need not explicitly create the block matrices to include active basis, so long as some algorithm is used to zero out the inactive basis. We do use a true projection matrix in our algorithm, but in contrast to most of the literature on THB-splines our projection is valid of endpoint interpolating B-splines which do not have uniform basis at the boundaries of the parametric domain. Our projection matrix $R^\ell$ for nonuniform B-spline basis is found in [203]. Critically, note that this construction is only valid for dyadic refinement. More general constructions are available, see for example [202, 204–207] Furthermore, Kiss, et al. describe the extension of their work to non-dyadic, and non-uniform spline spaces [208], here they suggest using standard knot insertion, along the lines of that shown in [207], and in the dyadic case, the effect is identical to the construction cached as matrix prolongation operator in [203], which is what is used in this paper.

Future developments could include extending the work here to a bi-orthogonal, $2^{nd}$ generation wavelet projection scheme, thereby keeping not only refinement but also restriction to linear time [203].

Those implementation details aside, THB-spline evaluation is the same across all variants. THB-spline curve evaluation proceeds as shown in algorithm 1. First there is the projection algorithm, which, as explained in [51], projects the control vertices, there called the basis function coefficients, to the finest level of detail. At the close of the algorithm, evaluation can proceed in the same way as for a B-spline basis for the finest level $N$, as also explained in [51].
Algorithm 1 THB-spline Evaluation

1: procedure PROJECT_ALL
2:  data:
3:  \( C^\ell \leftarrow \ell = 0, \ldots, N; \)
4:  \( R^\ell, X^\ell \leftarrow \ell = 0, \ldots, N; \)
5:  result: \( D^N \) so that \( v(x) = b^N(x)^T D^N \)
6:  \( D^0 = C^0 \)
7:  for \( \ell = 1 \) to \( N \):
8:  \( D^{\ell} = (I^{\ell} - X^{\ell}) R^{\ell} D^{\ell-1} + X^{\ell} C^{\ell}. \)
9:  return \( D^N \)

In this algorithm, we have

\( C \): control points

\( R \): projection (refinement) matrix

\( X \): diagonal matrix of active-inactive indices

We can stop here, as the algorithm above has performed the THB-spline truncation and has projected all the B-splines scaling coefficients (control vertices) to the finest level, where they can simply be evaluated as a standard B-spline.

A detailed explanation of the algorithm follows. This may be skipped if it is already clear above. Effectively in the explanation below, the basis are sent to zero by basis range/control vertex/bounds checking directly at evaluation time, as opposed to pre-computing matrices describing active and inactive basis/control vertices. This is simply a historical artifact of development at this point, and may be changed to reflect the above algorithm more closely at any time.

- Start with the coarsest level in the spline hierarchy.
- Only those control vertices which belong to basis functions which are completely enclosed within a bounding parametric interval for a given level are non-zero on that level.
- Typically the coarsest level is totally spanned by the active range so there is nothing to do but move to the next level.
- Denote this new level \( \ell \). On this level,
– Find the active basis/vertex indices for the bounds of the new prolonged level ($\ell$).

– For the vertices that actually originated at level ($\ell$), send to zero only those vertices or basis functions which have any portion active outside the bounds range for the curve at this level.

– Concurrently, prolong (project) the level ($\ell - 1$) control vertices to level ($\ell$) and send to zero any vertex associated with a basis function which is parameterized entirely within the active interval encapsulated by the bounds at the current level ($\ell$) in the hierarchy. This is a simple check (using basic basis function properties).

– Assuming nested spaces we need only ever examine one levels prolonged basis.

– On the coarsest level all basis remain active after this step

• Move to the next level and repeat the process. Project (prolongate) the vector (of control points in this case) on the current level to the next finer level via knot insertion (or, in actuality, its matrix equivalent projection operator). Simply matrix multiply the control vertices with the projection matrix. Set to zero the vertices from this level which are spanned (completely enclosed) by the active functions on the next level, and set to zero those functions from the next level which are, at any point, outside the active span there.

– This may not look like knot insertion, but it accomplishes the same thing.

– We use the wavelet projection matrices for endpoint interpolating B-splines as developed in [203].

– So my prolongation is actually a matrix vector multiplication.

– Other data, such as basis function coefficients, could be prolonged in this manner, but we focus on control points for reasons given above. (namely, so that the scheme is independent of parametric location)

• This concludes the projection. The procedure in effect acts separately and in fact oppositely on two objects per level of evaluation:

1. The vertices just projected there from the level below. (zero on this level’s active domain)

2. The vertices native to this level (zero outside this level’s active domain)

To be very explicit, we could also write down the relationship between a control vertex and its corresponding basis function, as pertains to determining whether a control vertex is active on a given interval.
Given a control vertex/basis function pair, the interval over which the basis function is non-zero is given by construction of a B-spline curve or surface itself. Let’s examine the details of this for a simple curve construction. The extension to higher dimensions is straightforward, if slightly more computationally involved due to the higher dimensional inclusion tests.

- First, given a basis function by its corresponding index, the parametric range over which it is active is given as

  \[ \text{active range} = [a, b] \]

  where

  \[ a = \tau(\text{basis index}) \]

  \[ b = \tau(\text{basis index} + k) \]

  Where \( \tau \) is the knot vector of the B-spline parameterization in this direction and the B-spline is of standard nonuniform type.

- Inversely, given a given parametric location, \( p \), the index range of all active basis functions are given as follows.

  \[ \text{active interval} = [\text{span} - p, \text{span} + 1] \]  \hspace{1cm} (10.6)

  Where \( p \) is the degree of the B-spline (curve, surface, etc.) in the direction being analyzed and the \( \text{span} \) is an integer denoting the \( i^{th} \) span between knots in the knot vector. See Piegl and Tiller [24] for details.

- These are the only basic algorithms required to find, given a particular basis function/control vertex index, whether it is active or inactive over a given parametric range. (Or to deduce a set of active functions).

- The extension to higher dimensions is a simple box inclusion test.

- A final complication is the need for parsimony — if overlapping bounds are specified, they should be unified.
**Important Properties Satisfied by THB-splines**

For THB-spline the following properties hold:

1. Stability of THB-spline basis under refinement is addressed in [44]. The basis is strongly stable with respect to the supremum norm. This means that the constants to be considered in the stability analysis of the basis do not depend on the number of hierarchical levels [209].

2. partition of unity, guaranteed by the truncated refinement scheme. (no need for special operations)

3. linear independence

4. local support

5. non-negativity

6. completeness

7. when refinement proceeds without deformation of the scaling functions (control vertices), the final geometry is identical with that of the original B-spline geometry, regardless of the number of refinement levels.

**Multilevel Editing**

As a first step towards full incorporation of the THB-splines into the automated hull form generation paradigm, they were used to add a bulbous bow to the bare ship hull form geometry. Later we will split the surface for greater topological freedom, but in this subsection, the result of this process is a single truncated hierarchical surface. The procedures incorporated here are derived from [51]. The main idea is an extension of the single level B-spline control mesh to the truncated hierarchical setting. Thanks to the convex hull property of the THB-splines, a multilevel control structure can be manipulated in much the same way as a traditional B-spline control vertex net.

In figure 10.1 an initial B-spline bare hull was first designed to constraints via standard form parameter design, and then a multilevel THB-spline representation of the surface was initialized from the old longitudinal control vertex net surface definition. Lofting as per Woodward, [158] was used to define a surface that exactly interpolates the original transverse curves, here shown in orange. As usual in Woodward’s lofting procedure,
Figure 10.1: Multilevel THB refined Bare Hull before deformation. Transverse curves shown in orange. (Displayed in Python’s Matplotlib library because Rhino does not support hierarchical splines, but only allows T-splines which we excluded from consideration in section 10.2 )
the longitudinal space curves were fitted with constraints such that they interpolate every transverse control vertex exactly. The resulting control vertices of the longitudinal curves then became the control vertex net of the B-spline surface. This B-spline surface then exactly interpolates the original transverse hull curves continuously along their lengths, but now the THB-splines offer the possibility of localized refinement. We simply ask that two levels of dyadic refinement be performed and that the active control vertices on the refined levels be advantageously located near the bow.

In the new THB-spline construction, the curves and surface from the B-spline construction become the initial level of the THB-spline surface curve and surface representation. Note that in this example the original surface was designed to have 7 control vertices in the transverse direction and 11 vertices in the longitudinal direction. Dyadic, nonuniform refinement of the knot vector in each principle direction would then split only each knot interval which lay on the interior of the endpoint interpolating knot vector. In practice, this means that the next surface level in the hierarchy would be 11 transverse vertices \( \times \) 19 longitudinal control vertices, though critically, only a fraction might be active at any time, as follows: We specify the refinement regions in the knot vector (parameterization) space. This naturally maps to active vertices in the spline space via the THB-evaluation procedure.

The surface was refined again and bounds were set such that there were many degrees of freedom near the nose of the hull, meanwhile the aft portions of the hull remained at the base level of refinement. Given the construction of the initial B-spline and bulbous bow, the exact refinement bounding parameters could be chosen to specify exactly enough degrees of freedom to allow, for instance, each refined vertex to map to an interior vertex on the bulbous bow, plus extra vertices to ensure C2 boundary conditions between the refined region and the original hull shape outside of the bulb area. In fact this choice was made to ensure enough degrees of freedom to fit the refined vertices to those of a bulbous bow and boundary, plus additional vertices higher up on the hull to enforce a nose tangency condition on the stem. For this initial study in multilevel editing, the nose tangency was thus enforced by direct, programmatic movement of those necessary vertices to enforce a flat nose. Later in this paper we will instead use THB-spline optimization techniques to properly enforce all original form parameters and have the freedom to experiment with boundary conditions or other continuity conditions at points of interest on the longitudinal curves, such as the stem boundary, bulb-bare hull interface, or even implicit conditions enforced between two neighboring curves.

The final hull form is shown in figure 10.2. This is a 3D view from port side looking at the half hull below the waterline.
Figure 10.2: Multilevel THB Hull after Bulbous Bow deformation with THB-spline refined control vertex net. Higher vertex counts denoted by blue circles denoting vertices at one higher level of refinement as compared to the magenta vertices aft. Please note that this single surface THB-spline will be refined further, especially the forward surface portions, and then projected to a B-spline representation at the highest level of detail and sent to Rhino for final display. See below for details and example hulls which have been further processed in this way.

As mentioned above, a bulbous bow had been separately defined via form parameter design. Furthermore, we have mentioned that multilevel THB-spline optimization can be successfully performed and is useful in this thesis. We develop multilevel optimization procedures below. Bulbous bow form parameter design is described in subsection 11.2, with final integration of the surfaces to come after that.

**Multilevel Optimization**

Assuming the original constraints on a THB-spline curve come from a surrounding program, detailed elsewhere in this thesis, the following are the basic steps of an adaptive optimization of the curve to meet the constraints in a fair manner consistent with work to date in automated ship hull form generation. These are summarized in figure 10.4.

- Initialize a THB-spline curve in that with enough degrees of freedom to satisfy linear algebraic constraints on the problem. (non-singular Hessian matrix for Newton's solver, etc.)

- Initialize a form parameter design curve optimization and fairness design problem of the same type as described elsewhere in this thesis.

  - Set a threshold for gradient driven refinement later. This could be determined absolutely, or
relative to some problem dependent measure.

- Set some reasonable number as the maximum number of vertices to be refined at once. This could be a percentage of the total number of degrees of freedom in the problem, or the number of degrees of freedom encapsulated by the control vertices, etc. The reason for this is that, even if the solver has one degree of freedom too few, several nearby vertex gradient components may show very high gradients as a result. Furthermore, one too few degrees of freedom might affect many constraints at once, though in theory, there is only “one constraint to many.” Therefore, even refining a few vertices in a localized area could result in sufficient gradient magnitude decreases across vertices that were not refined, and which might otherwise seem unrelated to a local problem and a commensurate local refinement.

- Run the algorithm as a standard B-spline optimization for at least one step.

  - THB projection is identical to B-spline evaluations since there is only one level.

  - Note we could incorporate THB levels all the way to the coarsest representation viable for the degree of the curve. This would be a full multi-grid geometric approach and may be the subject of future numerical trials. Other authors have conducted such studies, for example, [210].
• Use gradient thresholding to mark a set of vertices for refinement.
  
  – Since a control vertex has one associated basis function, all degrees of freedom encapsulated by that vertex are refined together.

• Institute THB-spline refinement.
  
  – For this program this means dyadic refinement of the knot vector as described especially simply by Stollnitz, et al., in their wavelet book [203].
    * Every interior segment of the knot vector is split in half during dyadic refinement of endpoint interpolating curves.
    * This leads to a simple refinement matrix projection operator given in the same work.
    * More efficient operators have been researched, especially so called “second generation schemes.” These are in general more complicated for endpoint interpolating wavelets than for more popular periodic wavelets. See [207] for more recent work in the endpoint interpolating case.
    * We project the entire curve to the finer level. Note that the entire basis of the curve is projected, but only those knots (and thus vertices) which are active within the parametric bounds supplied are actually used in the THB-spline evaluation procedure. Outside of the bounds of the finer level, the curve evaluation falls back to the lower resolution spline space.
  
  – The curve defined on this refined spline space is designated the child curve of the parent curve upon which we have been operating.
  
  – The dyadically refined curve has bounds designated such that exactly those vertices which were earmarked for refinement are now represented by a number of basis function — vertex pairs at the refined level.
  
  – There could be an option here to refine more generally around the area of high gradient, but for geometric fitting and shape optimization, this is thought to be subject to a trade off where excessive degrees of freedom lead to decreased shape quality. Therefore, we refine only the minimal amount.
  
  – Adapt the Lagrangian to the full dyadic increase in degrees of freedom. This becomes the child Lagrangian to the coarser curves Lagrangian.
• Projection
  
  – Project all vertices to the finest level in the THB-spline manner.
  – Note: (Projecting to a finer level is known as prolongation in multi-grid terminology)

• Evaluation
  
  – Evaluate the fine level Lagrangian there in the same manner as a standard B-spline curve.
  – After all, this is what the algorithm sees on the finest level. ¹

• Restriction
  
  – Update the fine vertices and project (restrict) them to the spline space hierarchy on the finer levels.
    (This is “restriction” in the language of multigrid).
  – Simple restriction averages the deformation of vertices which are not active on the fine level as represented by the deformation of vertices on the coarse level. This average is implicit in the restriction - we simply project the fine vertices back to their active, coarse, counterparts using the restriction operator developed by [203].

• Iterate to convergence, adapting the bounds of the curve levels via gradient thresholding if desired.

This procedure, including thresholding is primarily adapted from discussion in [201]. For our purposes, quite a low threshold of $ceiling = 1.e - 7$ was used to foster aggressive refinement, together with a low cap on the total number of vertices that could be refined, and a restriction to one or at most two levels of refinement. This was found to be sufficient for geometric optimization problems. Note that simpler formulations of multilevel editing without adaptive Lagrangian minimization can be found in [51]. This editing has been included in the hull form generation process as an automatic method for incorporating a bulbous bow in to the bare hull form.

Note that additional details could be included to aid in the generation of smooth geometry in the automated setting. For example, one might initialize the optimization problem on the coarse level as an approximate fitting problem, in order to produce a smooth starting solution at low resolution with a relatively high number of equality constraints. After a few iterations of approximate smoothing interpolation, gradient

¹We will later use the same fact of projection to transfer the THB surfaces to a B-spline drawing program, Rhino, for professional quality display
Initialize a THB-spline curve

Initialize the form parameter design problem

Run one or two iterations of FPD optimization

Mark vertices for refinement (gradient thresholding)

Refine designated vertices if they meet all criteria (e.g., restrictions on the number of nodes refined and number of levels allowable)

Run 5 iterations of THB-spline FPD optimization.

update vertices

converged?

no

yes

stop

Figure 10.4: THB-spline curve optimization algorithm flowchart
driven refinement and the imposition of equality constraints has a higher likelihood of success over, for example, heuristically having the program guess architectural decisions like the right number and placement of transverse curves. This is especially true when attempting to generate longitudinal space curves as one piece along the entire length of the hull form.

See figure 10.3 for preliminary results. Here the hull from the multi-level modeling section was altered to have 14 transverse curves in order to increase design accuracy in transition areas for the flat of side. A set of longitudinal curves were lofted using the optimization framework to exactly interpolate the transverse curves. Different colors denote different levels of refinement along the length of the longitudinal curves. Transverse curves are not refined here. Transverse surface curves would be refined, but this is not necessary with the underlying transverses from form parameter design, as these have already met all of their constraints and the localized level of detail seen by these transverse stations reduces the dependency on more global rules and relations. Later on, we will refine hull transverse curves in order to facilitate the addition of the bulbous bow.

The following are additional notes on the methodology:

- In this preliminary example, the solver takes one iteration using least squares approximation all of the transverse vertices (except the endpoints, which are fixed) to establish a smooth starting position.

- Then the gradients are used to pick out vertices for refinement. If gradients exceed an absolute bounds threshold (\[211\] for a discussion of thresholding choices) then a certain number of vertices are allowed to be added to a list to be refined.

- Refinement generates a new THB level for the surface via dyadic refinement in both the transverse, u, and longitudinal, v, directions. (Refinement of the surface control vertex mesh)

- Vertices are projected, in the THB-spline manner, one possible algorithm of which is given in algorithm 1, to the finest level, and the Lagrangian is evaluated in the standard way. (Note that any memoizing or caching that the solver does must be aware of the change in the size of the solver vectors and matrices. Code design dictates that each component should know how to “refine itself” to match what is required.)

- In practice, a limit is given to avoid excessive refinement and convergence is checked in the standard way for the curve solver demonstrated in \[212\] and shown elsewhere in this thesis.
Heuristic Findings

Some experimentation with constraints and with refinement has been carried out to get a better idea of the best ways to set up a multi-resolution longitudinal lofting solver.

- For optimization problems with equality constraints, typically only one or two levels of refinement might be needed in the worst cases.

- Almost any constraint can be used, because the curve is always evaluated by projecting to the finest level and solving the matrices there. However, some care is needed for constraints which are intended for particular vertices. This may not have a clear meaning unless imposed as a type of end condition, or similar.

Generating a THB-Surface From THB-curves

A procedure to incorporate THB-spline refined longitudinal curves into the THB-spline surface directly (with no additional optimization, and yet meeting the prescribed geometry) has been found and is used to generate THB-spline surfaces directly from the optimized THB-spline curves described above.

The procedure for this is as follows:

- Optimize the initial set of adaptive longitudinal curves as described above.

- Initialize the THB-spline surface with the coarse level control vertices. Call this level $\ell$

- The dyadic refinement procedure starts with the longitudinal direction. But we already have that refinement level from the optimization. Use the refined and deformed vertices from the longitudinal curves at the $(\ell + 1)$ level as the initial set of dyadically refined longitudinal curves at the $(\ell + 1)$ level.

- The next step in the basic refinement procedure is to cast transverse curves through all the intermediate control vertices at this proto-level $(\ell + 1)$. Be sure to cast through every single longitudinal vertex instead of just the refined ones.

- Dyadically refine these transverse curves to complete the transition to the true $(\ell + 1)$ level.

- This level now has automatically incorporated the finer details found in the $(\ell + 1)$ longitudinal curves.

- Extend the bounds at level $(\ell + 1)$ to cover the extra definition in the refined transverses as well.
Summary of THB-spline Findings

In the belief that “lofting can outperform smoothing” [78], using a multiresolution and topologically adaptable surface representation like the THB-splines, the program can design complex geometries into ship hulls directly by lofting, rather than by patching the surfaces together. Yet when the complexity and requirements dictate, lofting individual surface portions is a viable method as well. This is well facilitated by the framework shown here. This is just as well, because we will need to do so for the final stage of hull form generation to be shown in the next section.

The hope of multiresolution geometric design is to avoid trimming and intersecting parametric curves and surfaces altogether, sidestepping some of the costliest issues with CAD design when it is to be used in analysis. In fact, if the entire work-flow is to be automated and the analysis of the geometry is to really be incorporated into the design loop at the complex hull geometry level (and onward), then trimming, with its attendant analysis issues may need to be kept to a minimum [213].

Instead, in this thesis, we use the THB-spline optimization approach developed above, together with methodologies to be presented in the next section, to avoid the issues of trimming and splicing together surfaces in ways that might not be cross compatible with various modeling and analysis software. Instead, by projecting our surfaces to the finest B-spline level, we ensure maximum compatibility, while still exploiting some multiresolution capability for the actual curve solver computations.

Next, in the final program, and through a process of splitting surfaces, refining longitudinal curves, and fixing boundary conditions, we will demonstrate the automatic generation of feasible complex hulls. Furthermore, the full system will be integrated, with the constraint programming system ensuring feasible form parameters are used to initiate form parameter design, and finalizing design details through THB-spline methods. The details are given in the next section.
Chapter 11

Final Program

In section 7.1, An example of basic design of a sectional area curve was shown using automatic differentiation together with variational curve fairing and constraint satisfaction via method of Lagrange multipliers. In this section we demonstrate the robust nature of the system by generating a full complex ship hull form consisting of a bare hull and bulbous bow of an offshore supply vessel (OSV).

We will start with some design space, represented as a tuple of logical form parameter design variables together with an environment which maps from the variables to their respective interval values. The actual representation is from a design variable (form parameter) to a list of Python dictionaries, each representing a design subspace, as described in section 8.3.4. Each form parameter has an associated set of interval values, where each unique interval combines with all other unique intervals across the other parameters such that they all together represent the Cartesian product of all possible designs that have not been shown to be infeasible. This is the interval enclosure of the feasible design set, and is commonly referred to as “the design space”

Figure 11.1: Final output of the form parameter process: A multi-surface complex hull.
in this paper. Logical variables and starting values were given in sections 8.3.2 and 8.3.4 and generate
an offshore supply vessel baseline hull form, from which a designer might choose to do detail design, or a
hydrodynamics specialist might evaluate for motions and wave resistance, for example.

For our purposes though, we wish to generate only the geometry. To demonstrate the robustness of the
design procedures, designs selected from the design space randomly. In the subsections below we give the
details of how that is accomplished.

**Random Form Parameter Input Generation**

The relational rules system is able to generate sensible geometric form parameters from random inputs. Those
inputs go on to produce sensible geometry via the form parameter design. In our relational rules system a
random number between $[0, 1]$ is generated by the Python standard library. Meanwhile, a constraint from
the design space is selected and the natural interval parametrization of $[0, 1]$ is used to select a random thin
interval from the total feasible interval using the aforementioned random number. Note that every interval
number in the `ia` class has a natural map from $[0, 1]$ built in. 0 maps to the infimum and 1 maps to the
supremum, with a linear interpolation in between. If the design interval happened to be multi-valued, then
for simplicity we select the first design space from the list of all spaces and draw a random design from
there. If the design search were part of a full design space exploration, then we would want to be sure and
parameterize across the multi-valued interval.

This thin interval becomes a design choice for this particular form parameter. The arc consistency
algorithm [121] propagates this data throughout the constraint network, reducing other interval design
parameters via the simple interval arithmetic contractor algorithms [6] implemented via relational unification
[152] Note again for convenience that unification allows computation to flow both ways through a single
arithmetic interval expression, automatically taking care of the need to program forward and backward
contractors [173].

The design space may split on encountering a singularity via generalized interval arithmetic [6], but
thanks to the list processing of the declarative logic system [152], this is naturally accommodated by simply
adding another state to the list of states. In practice, though multiple design states may be produced in the
early goings, the design quickly settles into a single state. In fact, since the choices are random and thin, extra
states are typically eliminated quickly once created, since the change in this state induces arc-consistency to
take place over the rules it is associated with. Furthermore, if, say, two design spaces are created, the next choice of thin interval design parameter will often take other design space partition options out of the picture.

Using the above procedures, random feasible design can proceed as follows in the next subsections.

**First, select a feasible design at random from the design space**

In figure 11.2, the basic algorithm for generating a random, yet consistent and physically realizable, set of form parameters is given. This is the way in which we generate random designs in this thesis. The purpose of generating designs at random is simply to demonstrate the robustness of the form parameter consistency enforcement through constraint logic programming using interval arithmetic and local *hull consistency* contractor algorithms.

**Procedure:** The procedure for selecting random designs from the design space is given below. Note the procedure is performed once to generate form parameters for the overall hull form, and then again to determine form parameters for the bulbous bow. The generic procedure, which carries over unmodified to any rules base, is as follows:

1. Given a design space mapping of form parameters to interval values, first arc consistency is run to ensure a minimum (to the extent allowed) enclosure of the feasible domain, given all the initial design intervals and rules.

2. An algorithm loops over the form parameters Starting arbitrarily with any form parameter.
   - a random number generator produces a float between zero and one. This is used to select a “nearly-thin” interval from the initial interval for this form parameter. This is a random design choice.
     - Please note that this is done merely to convey the robustness of the constraint programming for ensuring feasible design spaces are available, and infeasible spaces are excluded, to useful extent.
   - Constraint propagation via arc-consistency is run to ensure the new design space contracts as much as possible given the new choice. The details were given in the paragraphs above explaining random form parameter input generation above. Here a quick caveat is in order, to stress that the algorithm is not infallible:
Figure 11.2: Basic algorithm using arc consistency for random, yet consistent, form parameter design selection. Note that during numerical experiments, the design space and rules could typically be searched over without very much, or any, backtracking needed.
– Due to the interval dependency problem, there is always a possibility that the random selection has chosen a portion of the interval that is actually infeasible.

– If so, arc-consistency will return a null valued design space. When this happens, back up to the prior design space and select another form parameter to narrow. This is “backtracking search” and is trivial to implement, given our language essentially records every change of state naturally. We simply save the state which existed before the infeasible choice, return to it, and pick a new location on the interval at random. In practice, this scenario almost never comes up, and if it does, it is dealt with quickly.

• At the end of one loop through all the design variables, check to see if any of them map to intervals which are not thin enough. That is, select some tolerance such a real valued curve solver can select any value with that interval range without worry that the value will cause the curve constraints to become infeasible with each other.

  – To be safe, set the size of this “thin interval tolerance” to the width of the tolerance of the curve solver itself.

  – Note that in practice, this tolerance can usually be a bit larger, but this is not rigorous.

  – If any of the design intervals are larger than the tolerance on their widths, then add them to a “redo” list.

3. If there are any form parameters on the re-do list, then loop over them as described above.

4. When all form parameter design intervals are thin, the algorithm is complete. We have arrived at a single consistent design.

5. End of Algorithm, Return a consistent set of Form Parameters.

Once the design space is maximally, i.e., idempotently [104], contracted by the procedures just outlined, we will have reduced our space to, approximately, a single design point, that is, a single design. This design just becomes standard input into the standard form parameter system of [52], with modifications introduced in section 3. We reiterate a summary of this procedure below for convenience. General details are given in the early sections of this thesis.

1in actuality, this will be a very thin interval — thinner than the tolerance of our optimization algorithm.
Form Parameter Design Based on Random Inputs

**Generate a set of hull form curves**  From the design form parameters selected above, a set of hull curves, namely the SAC, waterline, and center-plane curves, will be generated via form parameter design. For the bare hull, procedures essentially follow those of [52]. Note that the bulbous bow sectional area aids in forming the overall SAC curve here. Further, the bulbous bow SAC curve is generated as a using the random bulbous bow form parameters generated above, and the newly generated overall SAC curve for the hull. In this way, a natural fairing between bulb and hull can be controlled, since information from both hull and bulb is available. We will give details on how the control is accomplished in section 11.2.

**Generate a set of transverse hull definition curves**  From the curves of form, the transverse sections will be derived. Additional local constraints, which could be based off end condition curves in a full system, are applied to find nice “OSV-like” geometries. For instance:

- Vertical portions of sectional curves aft of the midship sections, at the deck.
- Horizontal portions of section curve fwd of the midship sections, at the keel.
- Local transition continuity on longitudinal lofting curves.

Since we are just demonstrating OSV-like hull forms, some of this detail has been relegated to the setup of the individual transverse curve solvers themselves. Some details are given in section 3.

**Generate a set of longitudinal space curves**  From these transverse station curves and suitable boundary conditions, a set of longitudinal space curves are generated via form parameter design. These are made to exactly interpolate the vertices of the transverse curves. The vertices of these longitudinal curves becomes the surface vertex network for surfaces that conform to the original transverse curves.

Once we have a bulbous bow, these longitudinal curves are represented by the truncated hierarchical splines and solved in an adaptive manner, as outlined in 10.5.

**Generate surfaces from the longitudinal space curves**

The control vertices of the (now segmented) longitudinal space curves are used to form the control vertex net of the surfaces. Since we follow the schema of Woodward [158], the longitudinal curve design scheme above ensures that the resulting surface exactly interpolates the transverse curves from step three, above.
Complex Hull Form With Bulbous Bow

The above approach is a basic outline of the “form parameter rules network to form parameter design to adaptive optimization” approach for design of a bare hull, but a bulbous bow will be added as well. The form parameter rules network will ensure compatible sizing between the bare hull and bulbous bow, and also responsible for ensuring the self-consistency of the form parameters to be used, including all global constraints and constraints used for the curves of form, Sectional area curve, waterline curve, and center plane keel profile curve (SAC, DWL, CPK, respectively).

Besides the bulbous bow form parameters, new inputs are the aft transom draft and the portion of the bulbous bow transverse fairing curve which will be allotted to fairing in the bulbous bow. With these new inputs, the following expanded procedures generate a complex hull:

1. Random generation of the bare hull form parameter design vector comes first. Arc-consistency, design space representation, and interval arithmetic methods are all used in the same way as before, for the bare hull alone.

2. A bulbous bow design is randomly selected from the bulbous bow design space using the same random design selection procedures outlined for the bare hull. As always in this thesis, this is to show the
Figure 11.4: Forward Fairness Curve: shown here, in gold (or lighter color if viewing in black and white). The relative longitudinal position of this curve is a “hyperparameter” of the program, as are the relative forward position (from bow to midships sections) of the other forward volume defining curve and the relative aft positions (from midships sections to stern) of the aft volume defining curves. The area of this particular curve, as with any transverse station curve, is derived from SAC. The following information sets this curve apart from other fore and aft volume curves: The fraction of this curve’s area which is contained in the lower “bulbous” or here, simply rounded, portion of this curve is another input to the system which can be chosen by the user. Since it will be known in advance, it is used to set the height of the bulbous bow fairing sectional area curve at this station. This is an additional curve which shows the volumetric distribution utilized in joining the bulbous bow to the bare hull. This curve is of great utility in the design and integration of a bulbous bow, as discussed in section 11.2.
robustness of the system against non-expert inputs.

3. Given a set of form parameters for both bare hull and bulbous bow, the bare hull solver begins as before.

- Using the bulbous bow form parameters, SAC is designed to automatically match the correct volume distribution and integrate the bulb with the hull. Further, sectional area at the termination of the transom is set by the designer. Both changes simply alter the starting and ending sectional area values (i.e. the “y-height”) of the sectional area curve at the endpoints.
  - Using the bulbous bow form parameters, especially length and sectional area, a bulbous bow sectional area curve is generated that uses the offset in the bare hull SAC and the bulb length.
  - Further, the bulbous bow SAC curve carries further aft of the bulbous bow, terminating with a height determined by the proportion of transverse bow fairing curve area set by input from the designer. (This could easily extended to be a rule based form parameter) Here, the program simply looks at the longitudinal station allotted to the bow fairness curve (set by hyperparameter), finds the commensurate section area from the bare hull SAC, and fixes the given proportion of that area to be the height of the bulbous bow sectional area transition curve at aft termination. See the figure 11.4 to review the placement of this curve, and its usage.
  - These curves, the bulbous bow sectional area curve and sectional area transition curve, reflect the reality that the transverse bow fairing curve controls the flow of the hull from the complex shape in the bow to the straighter sections aft.

- The center plane keel profile curve ending height is augmented to account for the transom. At this time a drop is simply entered as a single valued input parameter, though this could be an interval valued parameter with a commensurate rule in the future. Similarly, we could allow the deck line transverse width at aft endpoint to vary, but at present we have the deck run at full width all the way aft to the transom.

4. Next, the bulbous bow generation proceeds by form parameter design in the usual way.

5. Joint the bulbous bow and bare hull as described in subsection 11.3 and shown in chart 11.5.
Form Parameter Bulb Design

Bulbous bow generation proceeds as is typical of form parameter design systems. First, the bulbous bow rules from subsection 8.3.4 were added to the system and combined with the bare hull rules from subsection 8.3.2. Again random form parameter selection is employed, just as for the design selection of the bare hull form parameters, as described in section 11.2.

Secondly, the sectional area curve of the bulbous bow was generated during the solution of the overall SAC, as described in section 3. Note that for demonstration purposes, we use a constant sectional area through the chief bulb sections.

Next, form parameter curve solvers were used to generate a set of bulb curves which met the specification defined above. Form parameter design of the bulbous bow otherwise corresponds to the solution methods used for the bare hull, but with less detail (fewer constraints overall) as we are not as worried about local shape definition here. However, in order to fully integrate the bulbous bow with the hull, the sectional area curve must incorporate the bulbous bow such that the total area and longitudinal centroid of this curve accurately reflects the compound design. This was done organically by building the bulb design into the definition of the bare hull SAC to begin with. Here, the bulbous bow sectional area curve is determined during the solution of the bare hull sectional area curve, using a combination of user chosen parameters, listed below, and the actual bare hull forward fairness curve which is generated as part of the form parameter solution of the bare hull. The two specified input fractions are:

1. Bulb length fraction: This is the fraction of the distance from the bow of the ship to the hull forward fairness curve which is to be the length of the bulbous bow.

2. Bulb volume fraction: This is the fraction of the solved bare hull forward fairness curve area which is to be specified as the cross-sectional area of the bulbous bow.

Note that these could be made into rules as well. In current form, they are utilized to generate rules on bulb length and bulb sectional area which are then used in the bulbous bow form parameter rules generation algorithm, which acts exactly like that of the bare hull system to automatically and randomly generate consistent form parameter sets for bulbous bow form parameter design.

As stated above, once the consistent form parameter design inputs are known, FPD is solved as follows:

1. Curves of form are generated.
• Bulb SAC is generated during the solution of the overall ship hull sectional area curve.

• Vertical and horizontal center plane curves are generated based on form parameter bulb extreme bounds and area constraints, similar to the way in which bare hull rules were used to shape the waterline and center plane keel profile of the bare hull.

2. Transverse curves are generated using the sectional area and the vertical and horizontal plane curves.

3. A nose “transverse” curve is generated in order to generate a smooth cap on the bulb.

4. Longitudinal space curves are optimized to interpolate the control vertices of the transverse curves.

5. A B-spline surface is generated using the longitudinal control vertices as surface vertex net.

If the solver produced a bulbous bow in a coordinate system different to that of the bare hull, then it is transformed to match. By assumption, and via the construction outlined above, using the bare hull SAC, the bulbous bow is designed to start with the furthest forward point at the same location as that of the bare hull most forward bow location.

**THB Solver Split the Longitudinal Curves and Solve for Robust Local Details**

After the basic bare hull and bulb are generated, a second sweep of form parameter design takes place, only for the longitudinal sections, and this time using adaptive curves.\(^2\) This is a cleanup and preparation phase for the bulb to bare hull joining that will follow. By these more fine-grained optimization procedures of the longitudinal curves, final surface midship transitions, bulbous bow continuity, bow tangent, and flow from midship aft through the run can be controlled with maximum fidelity. The adaptive solution ensures that the consistency rules have only to worry about global constraints and broad range ramifications of local continuity conditions. The adaptive THB-spline solver makes sure that, e.g. the flat midsection is really flat, and that there is no issue interpolating all transverse curves, transitioning smoothly from one section of the hull to another, and meeting the water-line gracefully \(^3\) at stem and stern.

\(^2\) Note that we could have skipped the preliminary longitudinal B-spline optimization and “manual” THB-spline skinning of section 10.4 and gone straight to the adaptive longitudinal curve solver in preparation for creation of the split surface model. Yet the initial solutions are the product of software methodology experimentation, and the fact that there was some success in modeling the hull as a single surface was deemed interesting enough to include here, despite being an unnecessary step on the way to the multi-surface model. It’s almost certainly true, for instance, that if more freedom where allowed in the bulbous bow form, then we could forgo the decomposition of surfaces entirely and create a smooth blending shape, albeit with less local control over the form parameters.

\(^3\) Gracefully is a qualitative term; here meaning, “gracefully, for an automated system.”
Usage of THB splines to Assemble the Complex Hull

Now we use the THB optimization procedures to perform adaptive curve design and join the bare hull to the bulbous bow. The procedure is specified in chart 11.5. Details are described below.

Greater solver efficiency and robustness was found when switching to a split longitudinal solver and split surface creation process at this point. In other words, the single THB-spline surface will be split into multiple surfaces, via splitting of its underlying longitudinal space curves into three pieces corresponding to the forward, midship, and aft sections. The midship will be perfectly flat with change in longitudinal direction, and will not require further comment here, since it can be generated from its boundary curves, fore and aft, using simple linear curve segments.

The aft section will have its longitudinal curves solved again, to ensure the best fit to the transverse curves, and the forward hull will be split into lower and upper sections, and formed to match the boundary with the bulbous bow. Then the longitudinal curves of these split surfaces will also be solved afresh.

In this process, the aft longitudinal form parameters are given in table 11.1.

<table>
<thead>
<tr>
<th>Final Aft Longitudinal Form Parameter Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>α₁  Tangent angle into midship flat section</td>
</tr>
<tr>
<td>C₁  Curvature angle into midship flat section</td>
</tr>
<tr>
<td>qₓ₁ Transverse Aft midship transition curve vertex x-position</td>
</tr>
<tr>
<td>qₓ₂ Transverse Aft fairing curve vertex x-position</td>
</tr>
<tr>
<td>qᵧ₁ Transverse Aft midship transition curve vertex y-position</td>
</tr>
<tr>
<td>qᵧ₂ Transverse Aft fairing curve vertex y-position</td>
</tr>
<tr>
<td>qᶻ₁ Transverse Aft midship transition curve vertex z-position</td>
</tr>
<tr>
<td>qᶻ₂ Transverse Aft fairing curve vertex z-position</td>
</tr>
</tbody>
</table>

Table 11.1: Aft Longitudinal Curve Form Parameters

Similar form parameters would also be used for the forward hull, but since that surface will be split once more in order to best join with the bulbous bow. This process will be described next.
Split the Forward Surface: Final Integration Between Bare Hull and Bulbous Bow

The bulbous bow is integrated with the bare hull through a process of decomposing the forward hull surface into two surfaces, one “lower” surface which runs from the bulbous bow shape aft to midship, and one “upper” surface encompassing the remaining (upper) surface of the forward hull shape. For reference, in figure 11.1, the upper forward hull surface appears in blue while the lower forward hull surface appears in gold.

The bare hull is made to meet the bulbous bow in a fair manner by using fairness functional smoothing with exact interpolation of the control vertices defining the new bow shape. Additional continuity conditions assure a smooth meeting between bulb and upper and lower forward hull surfaces. Adaptive procedures aided in the automation of a smooth and fair interface between these geometries.
Boundary Conditions on the Forward Hull Split Surfaces

As might be inferred from figure 11.5, we need to specify boundary conditions on the surface connections by specifying continuity, tangential, and curvature conditions in some optimal manner on the longitudinal curves which will define those connections. In this work, the bulbous bow will not be altered. We philosophically consider its design requirements as coming from without — likely direct from hydrodynamics, and so attempt to deal with the bulb as it is given. Accordingly, the conditions on the upper and lower forward hull surface longitudinal curves must be specified to meet the bulb smoothly, and meet each other smoothly as well. In practice, the optimal methodology found is listed below.

Lower hull boundary conditions

For the lower forward hull surface we have the following boundary conditions:

- Tangential continuity from hull to bulb in the global x and y directions, i.e. the transverse-vertical plane of the vessel.

- Curvature continuity between upper and lower hull surface constrains vertex position in the global y and z directions, i.e. the vertical-longitudinal plane. Note also that in practice the best results were found by leaving the upper hull alone and adapting the lower hull to match.

- Constraints will be setup exactly the same on the bottom longitudinal curve defining the upper hull, ensuring (x,y,z) C0 continuity there. This constrains all vertices at the seam between upper and lower hull surfaces.

- Tangential continuity to midships is enforced in the examples shown, curvature continuity can easily be enforced if desired.

Upper hull boundary conditions

For the upper forward hull surface we have the following conditions:

- The driving condition on hull quality is that the longitudinal “flow” to the stem of the vessel should terminate with at least tangential continuity to the stem. In practice this dictates that the upper hull surface, that is, above the portion flowing to the bulbous bow, must be free to satisfy this stem constraint.
Other surface boundary constraints must be designed around this essential fact. Note critically that the transverse component of the vertices must be left free to adapt the longitudinal hull curves to the tangency condition at the stem and flow of the curve aft of that point.

- Constraints will be setup exactly the same on the bottom longitudinal curve of the upper hull as the upper longitudinal curve defining the lower hull, ensuring \( (x,y,z) \) C0 continuity at the boundary.

- Tangential continuity to midships is enforced in the examples shown, curvature continuity can easily be enforced if desired.

Form parameters used for the lower forward surface longitudinal curve solutions were generally as follows in table 11.2, seen here:

<table>
<thead>
<tr>
<th>Lower Forward Longitudinal Form Parameter Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha_1 )</td>
</tr>
<tr>
<td>( C_1 )</td>
</tr>
<tr>
<td>( q_{x1} )</td>
</tr>
<tr>
<td>( q_{x2} )</td>
</tr>
<tr>
<td>( q_{y1} )</td>
</tr>
<tr>
<td>( q_{y2} )</td>
</tr>
<tr>
<td>( q_{z1} )</td>
</tr>
<tr>
<td>( q_{z2} )</td>
</tr>
<tr>
<td>( \alpha_2 )</td>
</tr>
<tr>
<td>( C_2 )</td>
</tr>
</tbody>
</table>

Table 11.2: Lower Forward Hull, Longitudinal Curve Form Parameters

Form parameters used for the upper forward surface longitudinal curve solutions were generally as follows in table 11.3, seen here:

209
Upper Forward Hull, Longitudinal Form Parameter Inputs

$\alpha_1$  Tangent angle at to stem
$q_{x1}$  Transverse forward midship transition curve vertex x-position
$q_{x2}$  Transverse forward fairing curve vertex x-position
$q_{y1}$  Transverse forward midship transition curve vertex y-position
$q_{y2}$  Transverse forward fairing curve vertex y-position
$q_{z1}$  Transverse forward midship transition curve vertex z-position
$q_{z2}$  Transverse forward fairing curve vertex z-position
$\alpha_2$  Tangent angle at to midship
$C_2$  Curvature angle to midship (optional, usually disabled as smoother “flow” was observed)

Table 11.3: Upper Forward Longitudinal Curve Form Parameters

Note further that along the upper boundary of the lower surface, the curve must be solved for the same conditions as the lower boundary of the upper surface. In practice, some form parameters were manipulated in order to ensure the best transition between surfaces, given the constraints between bulb, upper, and lower hull surfaces.

In practice, the following concessions were made to transition between surfaces:

- For the upper forward hull surface, the 'x' vertices were free forward of the midship conditions, to maximize smoothness. Furthermore, vertex conditions set the “flow” to the stem. In order to ensure the tangent angles go to zero at the stem, the 'y' and 'z' vertices are fixed nearby at the first (tangent) position aft of the stem. These also dictate the conditions on the upper 'y' and 'z' first vertex positions of the lower surface, by continuity.

- For the lower forward hull surface, below the upper two vertex positions, the lower longitudinal curves are free to best flow into the bulbous bow. Accordingly, 'x' and 'y' conditions dictate C2 continuity into the bulbous bow boundary. Elsewhere, i.e. at the upper portion of the lower hull surface, first vertex position, the curves must match the 'y' and 'z' coordinates of the upper hull’s lower longitudinal 'y' and 'z' coordinates, in order to ensure continuity between the surfaces.
The above lists of boundary conditions are those we found to be of the greatest utility in creating well-behaved surfaces which replace the original bare hull forward surface and become, in addition, well-formed connections to the bulbous bow, and to each other, and the midship surface directly aft as well. 4

A Note for joining hull and bulbous bow — complex geometry creation

A procedure might clarify this process and the resulting geometry. Here, starting with the existing transverse curves and lines, the procedure is as follows:

- The portion of the hull aft of midships, shown in red in figure 11.1, is solved as one surface, and we will not comment further on it here, since it follows the original bare hull lines.

- The midship, gray surface in figure 11.7 (green in figure 11.1), is solved as a single surface following the original lines as well.

- The forward hull is more interesting. This surface is split into two surfaces and made to interface properly with the bulbous bow. The overview in figure 11.1 shows the upper forward hull surface in blue and the lower forward hull surface in gold. In figure 11.7, more details are shown. Here, many lessons were learned:

  - By splitting the forward surface into two pieces, topological freedom is gained. This is needed to fit a B-spline surface to a preexisting bulbous bow in such a way to allow for maximum control over the bulbous bow design. Figure 11.7 particularly shows the critical area as the joint between the bulbous bow in gold and the upper and lower forward hull surfaces in shades of blue.

  * Other methods could be imagined to work around these topological issues, primarily by hull deformation instead of meeting a preset number of bulb space curves, or curve vertices, but for we chose to use a form parameter generated bulb to maximize control of each surface in turn.

  * This necessitated a split in the forward primary hull surface (originally called the bare hull) to allow all surfaces to properly meet with no gaps or distortions.

  - The transverse forward fairness curve is critical in controlling the flow of volume to the nose of the vessel. The enclosed area between this curve and the center line is shown in figure 11.6.

  4Of course we cannot forget the additional conditions which must be specified — namely that the longitudinal curves must interpolate the transverse curve vertices as well.
Figure 11.6: Forward fairness curve design section.
Figure 11.7: Forward fairness curve components split and embedded in the forward hull form.
– It turns out that the lower transverse forward fairness curve should be designed, from the outset, to mimic the cross-sectional behavior of the bulbous bow, to simplify the final interface between the bulbous bow, lower forward primary surface, and upper forward primary surface.

In figure 11.7, the area swept out by this forward fairness curve is subdivided because the lower region of influence, here shown in green, actually has an additional area forcing function placed upon it as a result of these findings. Here we just want to highlight that constraint with this color.

– Carrying on, the lower forward hull surface should be made to mimic the bow tangency condition of the upper surface. This will ensure no volume distortions in the upper hull, and still allow the transition to the bulb to be fairly smooth, with little distortion of the lower primary forward surface either, as long as the transverse forward bow fairness curve is constrained to closely mimic (equal or exceed the cross-sectional area of) the section properties of the aft transverse slice of the bulbous bow.

- After the sub-procedures listed above, we next regenerate the bow curve to match the aft-most bulbous bow curve, while continuing above to serve as the stem of the bow.

- Boundary conditions listed above are enforced with form parameter curve design techniques.

- The standard form parameter design requirements for longitudinal curves — passing through the transverse control points are also prescribed by the form parameter design variational method.

Using the boundary conditions listed above, the goals of achieving a nice joining of the three surfaces: upper forward hull, lower forward hull, and bulbous bow, were achieved. Briefly then, the procedure is as follows:

1. Split the bare hull into three portions
   - Aft section
   - Midships
   - Forward

---

5 As opposed to the other way around, where the bottom of the upper surface could be made to mimic the upper edge of the lower primary forward surface.
Figure 11.8: Transverse curves. Bow curve at left. Transom curve to the right. Here we have elected to show the final bow profile, but note that there is a secondary design curve forming the outline of the aft portion of the bulb where it meets the stem.

2. Split the new forward section at the seventh transverse vertex, which matches the seven vertex transverse definition of our bulbous bow. The point is to have a matching number of transverse degrees of freedom between lower forward hull surface and the bulbous bow.

   - The simplicity of the stem forming the boundary of the new upper surface ensures a good fit between it and the previous stem curve.
   - Ensure consistent numbers of transverse vertices on all transverse curves of these new split surfaces.

3. Regenerate the lower surface bow curve so that it exactly matches that of the aft-most transverse curve of the bulbous bow.

4. Setup new longitudinal curve design optimization problems with boundary conditions as described above.

5. Generate the longitudinal space curves for both surfaces.

6. Skin the surfaces.

The resulting transverse curve network will now resemble that of figure 11.8. The transverse splittings are not shown here, but the procedure follows that of Piegl and Tiller [24] so that the curves are not altered. THB-spline formulations ensure adequate degrees of freedom throughout.
A summary of the steps is given in chart 11.5. By those procedures the bare hull and bulbous bow are connected with maximum curvature control, given that we are working with longitudinal curves and their boundary conditions for the final surface definition.
Figure 11.9: Final output of the form parameter process: A multi-surface complex hull shown cutaway about the center-plane line of symmetry.
Chapter 12

Results

Plots of hull forms shown here generally show only one half of the hull form, since, by symmetry, this is all that need be designed. Some figures show a full two sided hull however, if it was deemed a more effective presentation.

Plots of Generated OSV Hull Geometry

The first set of OSV ship hulls pictured here 12.3, 12.4, 12.5, 12.1, 12.2, 12.6, 12.7, 12.8, were generated by most of these procedures, but allowing for free position of the bulbous bow, and not enforcing a relationship between the forward transverse bow fairness curve and the bulb. So this first set of results for randomly generated OSVs used a similar but slightly simpler construction. These are denoted OSV 1 and OSV 2 in the results. These hull techniques were then modified to account for better integration with the overall section area curve, following the procedures outlined in subsection 3.4. Furthermore, superior fairing of the bulbous bow into the lower forward hull surface, and better connections between upper and lower forward hull surfaces were designed using processes described in section 11.2. Note that all hull forms make use of the surface splitting and constraint procedures,

Also note that the high resolution THB spline surface geometry was projected to identical B-spline surfaces, utilizing more control points. Then the B-spline geometry was exported into the CAGD package Rhino for better viewing. Section lines, waterlines, and buttock lines were created in Rhino as well. For these, we used Rhino’s fast projective methods to generate accurate slices of the B-spline surfaces.

In the figure 12.8, the bulbous bow transition to the primary hull has very little volume. This inward notch could be prevented by form parameter control of the lower transverse bow fairness curve as will be
Figure 12.1: Randomly Generated OSV 1: Rendered view of a hull generated before final bulbous bow design procedures were set. This hull had less control over the volume distribution forward due to a lack of inputs relating the bulb section area to forward fairness curve, and lack of input relating the bulb length to forward fairness station location.

Figure 12.2: Randomly Generated OSV 1: Rendered profile view. As shown here and in figure 12.1, the bulbous bow location is not well controlled and likely does not match the SAC curve very well. This is fixed by following the procedures outlined in subsections 3.4 and 3.4.
Figure 12.3: Randomly Generated OSV 1: Lines Plan 3D view

Figure 12.4: Randomly Generated OSV 1: Lines Plan elevation view
Figure 12.5: Randomly Generated OSV 1: Lines Plan section view
Furthermore, in this older hull form, the upper primary forward hull dispenses with its tangency condition (which enforces nose smoothness) as it approaches the lower primary forward surface. This approach to a water-tight fitting between the two surfaces induces distortions on the upper hull. A better way was found by instead forcing the lower primary forward hull surface to handle continuity with the upper surface and bulbous bow, supported by the bow fairness transverse curve. This is all detailed in the resulting figures below.

Figures 12.9 and 12.10 display “OSV 3.0”, which was used in order to experiment with continuity conditions between bulbous bow and the forward hull surface. Here more precise control of the forward hull shape is obtained.

Note the wavy buttock lines near the forward hull transition to bulbous bow in figure 12.11. This has to do with the complex interplay between the lower primary hull surface and the bulbous bow. These two surfaces must be joined with C2 continuity that requires matching vertices in the transverse plane. Meanwhile, the primary upper hull connection with the lower hull surface demands longitudinal and transverse vertex

---

1 Which, or course, are actually being defined by curves, or effectively, defined by one dimension at a time.
Figure 12.7: Randomly Generated OSV 2: lines plan section view. Bulb distortion easily corrected with modifications to boundary conditions. This will be demonstrated in other figures.

Figure 12.8: Randomly Generated OSV 2: Rendered view
Figure 12.9: Randomly Generated OSV 3: Isometric view of half body

Figure 12.10: Randomly Generated OSV 3: Rendered profile view
Figure 12.11: Randomly Generated OSV 3: Isometric lines view. Artifacts visible at transition to bulbous bow. Compare this with figure 12.17 where this is fixed at the expense of continuity to the bulb and figure 12.29 where it is finally fixed in a more satisfactory way, maintaining continuity everywhere by altering the blending properties of the bow fairing transverse curve.
coordinate matching, but not the vertical direction. At least at the triple intersection at the “top” of the bulb all coordinates must match. Fortunately at this point they are guaranteed to do so, since this is the location of the original surface splitting. Nevertheless, immediately down stream of this point, the vertices might be difficult to align in all the necessary planes at once. This issue, here manifested as the artifact in the buttock lines, can be remedied by incorporation of a parameter to control the transverse section area curve so that a bulbous lower portion of that curve closely mirrors, or slightly overestimates the sectional area of the bulbous bow at its endpoint (and interface with the lower primary surface). A properly controlled surface allows for a nice interface with all three surfaces, as shown in figure 12.29.

In figure 12.15 and 12.16 showing what we call “OSV 3.1,” we have modified “OSV 3” continuity properties from bulb to the lower hull to match more closely those of the upper surface, where there is a tangent condition at the bow edge. This comes at the expense of the smoothest possible continuity from lower bare hull surface to the bulbous bow itself. In future hull studies, one might impose a condition on the transverse bow fairness curve bulb area function to ensure that the lower surface had more natural continuity with the bulb by fixing this bulbous transverse area to be the same as the sectional area of the bulb at its aft edge.

Note that for the elevation view, for example figure 12.18, relatively constant spacing between buttock lines indicate similar tangential magnitudes [214]. This could be a constraint in the future.

In figure 12.21 an attempt was made to sacrifice continuity between bulbous bow and lower primary forward hull in order to more nicely connect the upper and lower hull surfaces there. This experiment did not pan out. The real issue is that the transverse bow fairness curve, which resides just aft of this problem, needs to more closely match volume distribution between its lower portions and the bulbous bow. This correction allows for better fitting between bulbous bow, the lower primary forward hull surface, and the upper primary forward hull surface.

Control of the surface as it flows from the midship flat section forward to the bow is effected by the transverse bow fairness blending curve. This curve represents an interface between several factors:

- “Flow” from the lower bow bare hull surface into the bulbous bow.

- The trade off in volume distribution in the upper portions of the bow vs the volume allotted to fairing the lower surface into the bulbous bow itself.

- The curve of the deck line.
Figure 12.12: Randomly Generated OSV 3: Bow view showing curvature to the deck
Figure 12.13: Randomly Generated OSV 3: Connection with Bulb
Figure 12.14: Randomly Generated OSV 3: Rendered side shot showing geometry from bulb running aft.

Figure 12.15: Randomly Generated OSV 3.1: Isometric view of half body
Figure 12.16: Randomly Generated OSV 3.1: Rendered profile view

Figure 12.17: Randomly Generated OSV 3.1: Isometric lines view. Note that this hull has issues with fairness to the bulb, as shown in figure 12.21. This is fixed in OSV 4, as shown in figure 12.29.

Figure 12.18: Randomly Generated OSV 3.1: Side profile showing buttock lines.
Figure 12.19: Randomly Generated OSV 3.1: 3D view showing geometry from bulb to midship

Figure 12.20: Randomly Generated OSV 3.1: Geometry from bulb to midship
Figure 12.21: Randomly Generated OSV 3.1: Hull bulb connection with sub-optimal continuity conditions specified
Figure 12.22: Randomly Generated OSV 3.1: Bow view showing curvature to the deck geometry from bulb to midship
Figure 12.23: Randomly Generated OSV 3: Transition to Bulb controlled by bow fairing curve geometry from bulb to midship. Compare with figure 12.24, which has improved tuning using alternate continuity given by the procedures detailed in section 11.3

See for example the following figures.

The bow fairing curve includes a bulbous lower portion which, when scaled to be of slightly more half (sectional) area than the aft-most portion of the bulb itself, maintains a smooth line into the bulb, while simultaneously retaining the ability to match exactly with the upper hull surface which includes a tangentially smooth approach to the bow. Managing these two conflicting demands is something that this system can automatically handle.

As you can see from the pictures of OSV 3, for example in figure 12.26, the bow transition curve is perhaps not doing an optimal job of controlling surface shape on approach to the bulbous bow. In fact there is too much area in the volume enclosed by the fairing surface, caused by a higher fraction of sectional area allotted to the bulb form in the transverse bow fairing curve. However, the transition to the upper forward
Figure 12.24: Randomly Generated OSV 4: Transition to Bulb controlled by bow fairing curve. Smoothness between lower bare hull and bulbous bow, and from lower bare hull to upper primary hull, is controlled by tuning the bow fairness transverse curve. Compare this with figure 12.23, where boundary conditions encourage curvature to bunch up in parts of the surface connections. Here is this figure, we achieve better control using the procedures outlined in section 11.3.
Figure 12.25: Randomly Generated OSV 4: Transition to Bulb controlled by bow fairing curve. Compare with figure 12.26
Figure 12.26: Randomly Generated OSV 3: Transition to Bulb controlled by bow fairing curve (to much bulbous area in the bow fairness transverse curve in OSV 3) Compare with figure 12.25
hull surface is handled well as shown in figure 12.27. Note that one outstanding constraint development task is to come up with rules or form parameter procedures could closely control and “smooth” the waterlines. Potentially some similar insight\textsuperscript{2} could be found to enable similar and parsimonious control of the waterlines.

\textsuperscript{2}Insight similar to that which allowed for control of the bulbous bow interface to the primary hull via the bow fairness section curve.
Figure 12.27: Randomly Generated OSV 3: Another shot of the transition to bulb controlled by the fairing curve. Transition from primary lower forward hull to upper forward hull near the bulbous bow has developed a notch. This can be tamed by adjusting the transverse bow fairness curve using procedural logic to ensure the fairness curve contains a small transverse bulb of sectional area comparable to, and slightly greater than, the bulbous bow section area. Note that the transition from the lower primary forward hull to the bulbous bow is quite smooth. Compare this with figure 12.28
Figure 12.28: Randomly Generated OSV 4: Transition to Bulb controlled by the transverse-bow-fairness-curve leaving degrees of freedom open to handle continuity between upper and lower primary hull surfaces. The transition from lower surface to bulb remains good. Compare this with figure 12.27.
Figure 12.29: Randomly Generated OSV 4: Buttock lines at the triple transition flow smoothly to the cusp thanks to form parameter control of the transverse bow fairness curve, and control of continuity at the bow, and also the control of continuity to the bulbous bow. Compare this with figure 12.11.
Figure 12.30: Randomly Generated OSV : Lines Plan. Yet another randomly generated variation of OSV lines. See also surface and curvature plot in figure 12.31
Figure 12.31: Randomly Generated OSV: Hull surface and Gaussian Curvature for the hull detailed with lines in figure 12.30 with curvature tolerance $t = [-0.05, 0.05]$
Figure 12.32: Randomly Generated OSV: Lines Plan. Randomly generated variation of OSV lines. Properties given in table 12.4 Surface and curvature shown in figure 12.33
Figure 12.33: Randomly Generated OSV: Hull surface and Gaussian Curvature for the hull given in figure 12.32. Here curvature tolerance = $[-0.05, 0.05]$.
Validation by Tables of Results

Next we list some randomly generated hull form parameter sets, their corresponding FPD values, and the percentage error between the two. This serves as validation for the methodology developed in the chapters presented earlier.

The constraint logic programming language enforces the consistency of the form parameter target values. These are then transformed into real hull form geometry by form parameter design. Target values are the initial consistent constraint parameters. These are generated by the constraint solver. Actual values denote the final values of the generated B-spline geometry. These values are results taken from form parameter designed geometry. (Note that a few corresponding geometric parameters are implicit. For instance, the correspondence between the area of the SAC curve and the volume of the actual hull. We do not query the hull for the actual volume, as it is an implicit property of the hull after geometry generation, and not a target solved for by the form parameter solvers directly. Instead we check that the SAC curve has an area that matches that with the prescribed area coming from the constraint solver.)

Percentage error is calculated as follows:

\[
\%\text{error} = \frac{\text{abs} (\text{mid} x_{clp} - x_{fpd})}{\text{mid} x_{clp}} \times 100\%
\]

Where \( x_{clp} \) denotes the midpoint of a thinned interval form parameter resulting from the constraint processing and random design selection, and \( x_{fpd} \) is the final value of the form parameter solved for by the nonlinear form parameter design solver. Such validation for example hulls, completely randomly generated, are given in table 12.1 and in table 12.2.

Caveats

- The bare hull center plane keel curve is designed and solved with the bulbous bow only approximately accounted for. The results stated here do no apply to the final center profile curve, but only to the bare hull center profile curve which is initially, and successfully solved for, as shown in the tables. This curve has a forward sweep corresponding to a bulbous bow, but the final outline of the center profile will only be fixed after bulbous bow generation via form parameter design. This final curve is slightly modified from the initial curve. Future work could ensure that there is no change. However, it may be
more important to generate high quality bulbs that are driven by hydrodynamic considerations. With that in mind, we have established in these numerical experiments that we have sufficient freedom to design the bulb as needed.

- As usual in form parameter design, volumetric properties of the final hull form are derived from the solved curves. These properties — such as final hull volumetric displacement and center of buoyancy, are not directly solved. Instead, they are implicitly approximated by the solution of the curve network.

In these tables we show the percentage error between the randomly selected form parameter design input vector, and the corresponding property of the final form parameter which came from the optimization process to make that property a reality in the ship’s design. This means, for example, that for the overall volumetric displacement, $\nabla$, of the ship, we compare the value given as input to the form parameter system by the constraint logic pre-processor, to the total area of the resulting sectional area curve for the complex hull.

Thus, we do not evaluate the goodness of fit of the actually geometry in such cases, but rather the ability of the solvers to match the constraints randomly generated by the constraint logic programming system. This is because the purpose of this dissertation is to demonstrate new methodologies supporting and enabling feasible, complex hull form design by the traditional form parameter method, using new methods exterior to already published methods, and not to improve the inner workings of the form parameter method itself. Even the THB-spline methodologies shown here have made no attempt to increase the accuracy of form parameter design itself, but to retain what accuracy is inherent in the method, and add to it the flexibility to more robustly and automatically combine geometry components into an entire complex hull form.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Form Parameter</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>%error 12.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>∇</td>
<td>Area of SAC curve, (Volume of Ship)</td>
<td>ia(33615.536,33615.541)</td>
<td>33615.539</td>
<td>0.0</td>
</tr>
<tr>
<td>LWL</td>
<td>Waterline Length</td>
<td>ia(111.807,111.813)</td>
<td>111.81</td>
<td>0.0</td>
</tr>
<tr>
<td>B</td>
<td>Ship Beam</td>
<td>ia(26.238,26.244)</td>
<td>26.241</td>
<td>0.0</td>
</tr>
<tr>
<td>D</td>
<td>Ship Draft</td>
<td>ia(17.182,17.188)</td>
<td>17.185</td>
<td>0.0</td>
</tr>
<tr>
<td>A_M</td>
<td>Midship Section Area</td>
<td>ia(428.246,428.252)</td>
<td>428.249</td>
<td>0.0</td>
</tr>
<tr>
<td>A_C</td>
<td>Center-plane Area</td>
<td>ia(1545.681,1545.687)</td>
<td>1545.684</td>
<td>0.0</td>
</tr>
<tr>
<td>A_W</td>
<td>Waterline Area</td>
<td>ia(2650.703,2650.709)</td>
<td>2650.706</td>
<td>-0.0</td>
</tr>
<tr>
<td>LCB</td>
<td>Longitudinal Center of Buoyancy</td>
<td>ia(55.266,55.272)</td>
<td>55.269</td>
<td>0.0</td>
</tr>
<tr>
<td>C_B</td>
<td>Block Coefficient</td>
<td>ia(0.667,0.667)</td>
<td>0.667</td>
<td>-0.008</td>
</tr>
<tr>
<td>C_P</td>
<td>Prismatic Coefficient</td>
<td>ia(0.702,0.702)</td>
<td>0.702</td>
<td>-0.008</td>
</tr>
<tr>
<td>C_M</td>
<td>Midship Area coefficient</td>
<td>ia(0.949,0.95)</td>
<td>0.95</td>
<td>-0.0</td>
</tr>
<tr>
<td>CWL</td>
<td>Waterline area coefficient</td>
<td>ia(0.903,0.904)</td>
<td>0.903</td>
<td>-0.0</td>
</tr>
<tr>
<td>LF_WL</td>
<td>Length ocenterf Flat of Waterline</td>
<td>ia(11.83,11.836)</td>
<td>11.833</td>
<td>0.0</td>
</tr>
<tr>
<td>CPK</td>
<td>Center-plane Keel Profile area coefficient</td>
<td>ia(0.804,0.805)</td>
<td>0.804</td>
<td>-0.0</td>
</tr>
<tr>
<td>LF_CPK</td>
<td>Length of the Center-plane Keel Profile</td>
<td>ia(8.94,8.946)</td>
<td>8.943</td>
<td>0.0</td>
</tr>
<tr>
<td>LF_CPK</td>
<td>Area of the Center-plane Keel Profile</td>
<td>ia(8.94,8.946)</td>
<td>8.943</td>
<td>0.0</td>
</tr>
<tr>
<td>LF_SAC</td>
<td>Length of Flat of SAC</td>
<td>ia(8.899,8.905)</td>
<td>8.902</td>
<td>0.0</td>
</tr>
<tr>
<td>DLR</td>
<td>Displacement to Length Ratio</td>
<td>ia(0.024,0.024)</td>
<td>0.024</td>
<td>-0.0</td>
</tr>
<tr>
<td>∇^Bulb</td>
<td>Area of Bulbous Bow</td>
<td>ia(78.078,78.082)</td>
<td>78.081</td>
<td>0.002</td>
</tr>
<tr>
<td>C_A_{Mbulb}</td>
<td>bulb sectional area maximum</td>
<td>ia(0.535,0.535)</td>
<td>0.535</td>
<td>0.002</td>
</tr>
<tr>
<td>C_A_{Wbulb}</td>
<td>Water-plane Bulb Coefficient</td>
<td>ia(0.964,0.964)</td>
<td>0.964</td>
<td>0.024</td>
</tr>
<tr>
<td>C_C_{bulb}</td>
<td>Center-plane Bulb Coefficient</td>
<td>ia(0.892,0.892)</td>
<td>0.892</td>
<td>0.001</td>
</tr>
<tr>
<td>C_B_{bulb}</td>
<td>Bulb Block Coefficient</td>
<td>ia(0.535,0.535)</td>
<td>0.535</td>
<td>0.02</td>
</tr>
<tr>
<td>C_P_{bulb}</td>
<td>Bulb Prismatic Coefficient</td>
<td>ia(1.0,1.0)</td>
<td>1.0</td>
<td>0.02</td>
</tr>
<tr>
<td>C_{bb}</td>
<td>Breadth Ratio</td>
<td>ia(0.239,0.239)</td>
<td>0.239</td>
<td>-0.01</td>
</tr>
<tr>
<td>C_{lpr}</td>
<td>Ship Length to bulb length ratio</td>
<td>ia(0.038,0.038)</td>
<td>0.038</td>
<td>-0.011</td>
</tr>
<tr>
<td>C_{zb}</td>
<td>Ship Draft to Bulb height Ratio</td>
<td>ia(0.314,0.314)</td>
<td>0.314</td>
<td>0.0</td>
</tr>
<tr>
<td>C_{abt}</td>
<td>Midship to Mid bulb area ratio</td>
<td>ia(0.042,0.042)</td>
<td>0.042</td>
<td>-0.009</td>
</tr>
<tr>
<td>C_{abl}</td>
<td>Center-plane Ratio</td>
<td>ia(0.013,0.013)</td>
<td>0.013</td>
<td>-0.009</td>
</tr>
<tr>
<td>C_{vpr}</td>
<td>Displacement Ratio</td>
<td>ia(0.002,0.002)</td>
<td>0.002</td>
<td>-0.01</td>
</tr>
<tr>
<td>B_{bulb}</td>
<td>Beam of Bulbous Bow</td>
<td>ia(6.28,6.281)</td>
<td>6.28</td>
<td>0.0</td>
</tr>
<tr>
<td>D_{bulb}</td>
<td>Draft of Bulbous Bow</td>
<td>ia(5.401,5.401)</td>
<td>5.401</td>
<td>0.0</td>
</tr>
<tr>
<td>L_{bulb}</td>
<td>Length of Bulbous Bow</td>
<td>ia(4.301,4.301)</td>
<td>4.301</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 12.1: Hull Design Randomly Generated Hull Form parameters
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Form Parameter</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>% Δ Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>∇</td>
<td>Area of SAC curve, (Volume of Ship)</td>
<td>ia(34709.168,34709.171)</td>
<td>34709.17</td>
<td>0.0</td>
</tr>
<tr>
<td>L&lt;sub&gt;WL&lt;/sub&gt;</td>
<td>Waterline Length</td>
<td>ia(127.544,127.55)</td>
<td>127.547</td>
<td>0.0</td>
</tr>
<tr>
<td>B</td>
<td>Ship Beam</td>
<td>ia(27.34,27.346)</td>
<td>27.343</td>
<td>0.0</td>
</tr>
<tr>
<td>D</td>
<td>Ship Draft</td>
<td>ia(15.644,15.65)</td>
<td>15.647</td>
<td>0.0</td>
</tr>
<tr>
<td>A&lt;sub&gt;M&lt;/sub&gt;</td>
<td>Midship Section Area</td>
<td>ia(399.703,399.709)</td>
<td>399.706</td>
<td>-0.0</td>
</tr>
<tr>
<td>A&lt;sub&gt;CP&lt;/sub&gt;</td>
<td>Center-plane Area</td>
<td>ia(1593.515,1593.521)</td>
<td>1593.518</td>
<td>0.0</td>
</tr>
<tr>
<td>A&lt;sub&gt;WP&lt;/sub&gt;</td>
<td>Waterline Area</td>
<td>ia(3228.355,3228.361)</td>
<td>3228.358</td>
<td>-0.0</td>
</tr>
<tr>
<td>LCB</td>
<td>Longitudinal Center of Buoyancy</td>
<td>ia(63.348,63.354)</td>
<td>63.351</td>
<td>0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;B&lt;/sub&gt;</td>
<td>Block Coefficient</td>
<td>ia(0.636,0.636)</td>
<td>0.636</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;P&lt;/sub&gt;</td>
<td>Prismatic Coefficient</td>
<td>ia(0.68,0.681)</td>
<td>0.681</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;M&lt;/sub&gt;</td>
<td>Midship Area coefficient</td>
<td>ia(0.934,0.935)</td>
<td>0.934</td>
<td>-0.0</td>
</tr>
<tr>
<td>CWL</td>
<td>Waterline area coefficient</td>
<td>ia(0.926,0.926)</td>
<td>0.926</td>
<td>-0.0</td>
</tr>
<tr>
<td>LF&lt;sub&gt;WL&lt;/sub&gt;</td>
<td>Length of Flat of Waterline</td>
<td>ia(13.456,13.462)</td>
<td>13.459</td>
<td>-0.0</td>
</tr>
<tr>
<td>CPK</td>
<td>Center-plane Keel Profile area coefficient</td>
<td>ia(0.798,0.799)</td>
<td>0.798</td>
<td>-0.0</td>
</tr>
<tr>
<td>LF&lt;sub&gt;CPK&lt;/sub&gt;</td>
<td>Length of the Center-plane Keel Profile</td>
<td>ia(10.817,10.823)</td>
<td>10.82</td>
<td>0.0</td>
</tr>
<tr>
<td>LF&lt;sub&gt;CWP&lt;/sub&gt;</td>
<td>Area of the Center-plane Keel Profile</td>
<td>ia(10.817,10.823)</td>
<td>10.82</td>
<td>0.0</td>
</tr>
<tr>
<td>LF&lt;sub&gt;SAC&lt;/sub&gt;</td>
<td>Length of Flat of SAC</td>
<td>ia(10.022,10.028)</td>
<td>10.025</td>
<td>-0.0</td>
</tr>
<tr>
<td>DLR</td>
<td>Displacement to Length Ratio</td>
<td>ia(0.017,0.017)</td>
<td>0.017</td>
<td>-0.0</td>
</tr>
<tr>
<td>∇&lt;sub&gt;Bulb&lt;/sub&gt;</td>
<td>Area of Bulbous Bow</td>
<td>ia(120.495,120.496)</td>
<td>120.513</td>
<td>0.015</td>
</tr>
<tr>
<td>C&lt;sub&gt;AMbulb&lt;/sub&gt;</td>
<td>bulb sectional area maximum</td>
<td>ia(0.911,0.912)</td>
<td>0.911</td>
<td>-0.003</td>
</tr>
<tr>
<td>C&lt;sub&gt;WPbulb&lt;/sub&gt;</td>
<td>Water-plane Bulb Coefficient</td>
<td>ia(0.73,0.73)</td>
<td>0.73</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;CPbulb&lt;/sub&gt;</td>
<td>Center-plane Bulb Coefficient</td>
<td>ia(0.715,0.715)</td>
<td>0.715</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;bulb&lt;/sub&gt;</td>
<td>Bulb Block Coefficient</td>
<td>ia(0.911,0.911)</td>
<td>0.911</td>
<td>0.016</td>
</tr>
<tr>
<td>C&lt;sub&gt;Pbulb&lt;/sub&gt;</td>
<td>Bulb Prismatic Coefficient</td>
<td>ia(1.0,1.0)</td>
<td>1.0</td>
<td>0.002</td>
</tr>
<tr>
<td>C&lt;sub&gt;b bulb&lt;/sub&gt;</td>
<td>Breadth Ratio</td>
<td>ia(0.234,0.235)</td>
<td>0.234</td>
<td>-0.003</td>
</tr>
<tr>
<td>C&lt;sub&gt;lpr&lt;/sub&gt;</td>
<td>Ship Length to bulb length ratio</td>
<td>ia(0.034,0.034)</td>
<td>0.034</td>
<td>-0.009</td>
</tr>
<tr>
<td>C&lt;sub&gt;zh&lt;/sub&gt;</td>
<td>Ship Draft to Bulb height Ratio</td>
<td>ia(0.305,0.305)</td>
<td>0.305</td>
<td>0.001</td>
</tr>
<tr>
<td>C&lt;sub&gt;abt&lt;/sub&gt;</td>
<td>Midship to Mid bulb area ratio</td>
<td>ia(0.07,0.07)</td>
<td>0.07</td>
<td>0.007</td>
</tr>
<tr>
<td>C&lt;sub&gt;abl&lt;/sub&gt;</td>
<td>Center-plane Ratio</td>
<td>ia(0.009,0.009)</td>
<td>0.009</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;vpr&lt;/sub&gt;</td>
<td>Displacement Ratio</td>
<td>ia(0.003,0.003)</td>
<td>0.003</td>
<td>0.005</td>
</tr>
<tr>
<td>B&lt;sub&gt;bulb&lt;/sub&gt;</td>
<td>Beam of Bulbous Bow</td>
<td>ia(6.412,6.412)</td>
<td>6.412</td>
<td>0.0</td>
</tr>
<tr>
<td>D&lt;sub&gt;bulb&lt;/sub&gt;</td>
<td>Draft of Bulbous Bow</td>
<td>ia(4.77,4.77)</td>
<td>4.77</td>
<td>0.0</td>
</tr>
<tr>
<td>L&lt;sub&gt;bulb&lt;/sub&gt;</td>
<td>Length of Bulbous Bow</td>
<td>ia(4.323,4.323)</td>
<td>4.323</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 12.2: Hull Design Randomly Generated Hull Form parameters
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Form Parameter</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>% error</th>
<th>∆</th>
</tr>
</thead>
<tbody>
<tr>
<td>∇</td>
<td>Area of SAC curve, (Volume of Ship)</td>
<td>ia(25916.122,25916.128)</td>
<td>25916.125</td>
<td>0.0</td>
<td>12.1</td>
</tr>
<tr>
<td>L_WL</td>
<td>Waterline Length</td>
<td>ia(110.329,110.335)</td>
<td>110.332</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Ship Beam</td>
<td>ia(25.014,25.02)</td>
<td>25.017</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Ship Draft</td>
<td>ia(15.122,15.127)</td>
<td>15.124</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>A_M</td>
<td>Midship Section Area</td>
<td>ia(343.838,343.844)</td>
<td>343.841</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>A_CP</td>
<td>Center-plane Area</td>
<td>ia(1317.874,1317.88)</td>
<td>1317.877</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>A_WP</td>
<td>Waterline Area</td>
<td>ia(2584.61,2584.616)</td>
<td>2584.613</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>LCB</td>
<td>Longitudinal Center of Buoyancy</td>
<td>ia(55.447,55.453)</td>
<td>55.45</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>C_B</td>
<td>Block Coefficient</td>
<td>ia(0.621,0.621)</td>
<td>0.621</td>
<td>-0.015</td>
<td></td>
</tr>
<tr>
<td>C_P</td>
<td>Prismatic Coefficient</td>
<td>ia(0.683,0.684)</td>
<td>0.683</td>
<td>-0.015</td>
<td></td>
</tr>
<tr>
<td>C_M</td>
<td>Midship Area coefficient</td>
<td>ia(0.909,0.909)</td>
<td>0.909</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>CWL</td>
<td>Waterline area coefficient</td>
<td>ia(0.936,0.937)</td>
<td>0.936</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>L_FWL</td>
<td>Length of Flat of Waterline</td>
<td>ia(13.098,13.104)</td>
<td>13.101</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>CPK</td>
<td>Center-plane Keel Profile area coefficient</td>
<td>ia(0.79,0.79)</td>
<td>0.79</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>L_FCPK</td>
<td>Length of the Center-plane Keel Profile</td>
<td>ia(11.05,11.056)</td>
<td>11.053</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>L_FCWP</td>
<td>Area of the Center-plane Keel Profile</td>
<td>ia(11.05,11.056)</td>
<td>11.053</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>L_FSAC</td>
<td>Length of Flat of SAC</td>
<td>ia(10.301,10.307)</td>
<td>10.304</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>DLR</td>
<td>Displacement to Length Ratio</td>
<td>ia(0.019,0.019)</td>
<td>0.019</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>V_Bulb</td>
<td>Area of Bulbous Bow</td>
<td>ia(47.22,47.22)</td>
<td>47.22</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>C_Ambulb</td>
<td>bulb sectional area maximum</td>
<td>ia(0.482,0.482)</td>
<td>0.482</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>C_AWPbulb</td>
<td>Water-plane Bulb Coefficient</td>
<td>ia(0.739,0.739)</td>
<td>0.739</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>C_CPbulb</td>
<td>Center-plane Bulb Coefficient</td>
<td>ia(0.549,0.549)</td>
<td>0.549</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>C_Bbulb</td>
<td>Bulb Block Coefficient</td>
<td>ia(0.482,0.482)</td>
<td>0.482</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>C_Pbulb</td>
<td>Bulb Prismatic Coefficient</td>
<td>ia(1.0,1.0)</td>
<td>1.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>C_b</td>
<td>Breadth Ratio</td>
<td>ia(0.201,0.201)</td>
<td>0.201</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>C_lpr</td>
<td>Ship Length to bulb length ratio</td>
<td>ia(0.039,0.039)</td>
<td>0.039</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>C_zb</td>
<td>Ship Draft to Bulb height Ratio</td>
<td>ia(0.301,0.301)</td>
<td>0.301</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>C_abt</td>
<td>Midship to Mid bulb area ratio</td>
<td>ia(0.032,0.032)</td>
<td>0.032</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>C_abl</td>
<td>Center-plane Ratio</td>
<td>ia(0.008,0.008)</td>
<td>0.008</td>
<td>-0.0</td>
<td></td>
</tr>
<tr>
<td>C_vpr</td>
<td>Displacement Ratio</td>
<td>ia(0.002,0.002)</td>
<td>0.002</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>B_bulb</td>
<td>Beam of Bulbous Bow</td>
<td>ia(5.024,5.024)</td>
<td>5.024</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>D_bulb</td>
<td>Draft of Bulbous Bow</td>
<td>ia(4.549,4.549)</td>
<td>4.549</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>L_bulb</td>
<td>Length of Bulbous Bow</td>
<td>ia(4.285,4.285)</td>
<td>4.285</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 12.3: Hull Design Randomly Generated Hull Form parameters
### Ship Hull Design Random Form parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Form Parameter</th>
<th>Value (1)</th>
<th>Value (2)</th>
<th>% error</th>
</tr>
</thead>
<tbody>
<tr>
<td>∇</td>
<td>Area of SAC curve, (Volume of Ship)</td>
<td>(34788.551, 34788.556)</td>
<td>34788.554</td>
<td>0.0</td>
</tr>
<tr>
<td>L&lt;sub&gt;WL&lt;/sub&gt;</td>
<td>Waterline Length</td>
<td>(125.779, 125.785)</td>
<td>125.782</td>
<td>0.0</td>
</tr>
<tr>
<td>B</td>
<td>Ship Beam</td>
<td>(29.915, 29.921)</td>
<td>29.918</td>
<td>0.0</td>
</tr>
<tr>
<td>D</td>
<td>Ship Draft</td>
<td>(15.0, 15.004)</td>
<td>15.002</td>
<td>0.0</td>
</tr>
<tr>
<td>A&lt;sub&gt;M&lt;/sub&gt;</td>
<td>Midship Section Area</td>
<td>(406.734, 406.74)</td>
<td>406.737</td>
<td>0.0</td>
</tr>
<tr>
<td>A&lt;sub&gt;CP&lt;/sub&gt;</td>
<td>Center-plane Area</td>
<td>(1583.563, 1583.569)</td>
<td>1583.566</td>
<td>0.0</td>
</tr>
<tr>
<td>A&lt;sub&gt;WP&lt;/sub&gt;</td>
<td>Waterline Area</td>
<td>(3514.874, 3514.88)</td>
<td>3514.877</td>
<td>-0.0</td>
</tr>
<tr>
<td>LCB</td>
<td>Longitudinal Center of Buoyancy</td>
<td>(65.27, 65.276)</td>
<td>65.273</td>
<td>0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;B&lt;/sub&gt;</td>
<td>Block Coefficient</td>
<td>(0.616, 0.616)</td>
<td>0.616</td>
<td>-0.011</td>
</tr>
<tr>
<td>C&lt;sub&gt;P&lt;/sub&gt;</td>
<td>Prismatic Coefficient</td>
<td>(0.68, 0.68)</td>
<td>0.68</td>
<td>-0.016</td>
</tr>
<tr>
<td>C&lt;sub&gt;M&lt;/sub&gt;</td>
<td>Midship Area coefficient</td>
<td>(0.906, 0.906)</td>
<td>0.906</td>
<td>-0.01</td>
</tr>
<tr>
<td>CWL</td>
<td>Waterline area coefficient</td>
<td>(0.934, 0.934)</td>
<td>0.934</td>
<td>-0.0</td>
</tr>
<tr>
<td>L&lt;sub&gt;FWL&lt;/sub&gt;</td>
<td>Length of Flat of Waterline</td>
<td>(18.191, 18.197)</td>
<td>18.194</td>
<td>-0.0</td>
</tr>
<tr>
<td>CPK</td>
<td>Center-plane Keel Profile area coefficient</td>
<td>(0.839, 0.839)</td>
<td>0.839</td>
<td>-0.0</td>
</tr>
<tr>
<td>L&lt;sub&gt;FCPK&lt;/sub&gt;</td>
<td>Length of Center-plane Keel Profile</td>
<td>(16.416, 16.422)</td>
<td>16.419</td>
<td>-0.0</td>
</tr>
<tr>
<td>L&lt;sub&gt;FSAC&lt;/sub&gt;</td>
<td>Length of Flat of SAC</td>
<td>(14.98, 14.986)</td>
<td>14.983</td>
<td>-0.0</td>
</tr>
<tr>
<td>DLR</td>
<td>Displacement to Length Ratio</td>
<td>(0.017, 0.017)</td>
<td>0.017</td>
<td>-0.0</td>
</tr>
<tr>
<td>∇&lt;sub&gt;Bulb&lt;/sub&gt;</td>
<td>Area of Bulbous Bow</td>
<td>(57.835, 57.863)</td>
<td>57.861</td>
<td>0.02</td>
</tr>
<tr>
<td>C&lt;sub&gt;AMbulb&lt;/sub&gt;</td>
<td>Bulb sectional area maximum</td>
<td>(0.545, 0.546)</td>
<td>0.546</td>
<td>0.006</td>
</tr>
<tr>
<td>C&lt;sub&gt;AWPbulb&lt;/sub&gt;</td>
<td>Water-plane Bulb Coefficient</td>
<td>(0.907, 0.907)</td>
<td>0.907</td>
<td>-0.001</td>
</tr>
<tr>
<td>C&lt;sub&gt;CPrbulb&lt;/sub&gt;</td>
<td>Center-plane Bulb Coefficient</td>
<td>(0.942, 0.942)</td>
<td>0.942</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;Bbulb&lt;/sub&gt;</td>
<td>Bulb Block Coefficient</td>
<td>(0.545, 0.546)</td>
<td>0.546</td>
<td>0.02</td>
</tr>
<tr>
<td>C&lt;sub&gt;Prbulb&lt;/sub&gt;</td>
<td>Bulb Prismatic Coefficient</td>
<td>(0.999, 1.0)</td>
<td>1.0</td>
<td>0.064</td>
</tr>
<tr>
<td>C&lt;sub&gt;bb&lt;/sub&gt;</td>
<td>Breadth Ratio</td>
<td>(0.162, 0.162)</td>
<td>0.162</td>
<td>-0.039</td>
</tr>
<tr>
<td>C&lt;sub&gt;lpr&lt;/sub&gt;</td>
<td>Ship Length to bulb length ratio</td>
<td>(0.04, 0.04)</td>
<td>0.04</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;zbb&lt;/sub&gt;</td>
<td>Ship Draft to Bulb height Ratio</td>
<td>(0.293, 0.293)</td>
<td>0.293</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;abt&lt;/sub&gt;</td>
<td>Midship to Mid bulb area ratio</td>
<td>(0.029, 0.029)</td>
<td>0.029</td>
<td>0.001</td>
</tr>
<tr>
<td>C&lt;sub&gt;abl&lt;/sub&gt;</td>
<td>Center-plane Ratio</td>
<td>(0.013, 0.013)</td>
<td>0.013</td>
<td>-0.0</td>
</tr>
<tr>
<td>C&lt;sub&gt;vpr&lt;/sub&gt;</td>
<td>Displacement Ratio</td>
<td>(0.002, 0.002)</td>
<td>0.002</td>
<td>0.02</td>
</tr>
<tr>
<td>B&lt;sub&gt;bulb&lt;/sub&gt;</td>
<td>Beam of Bulbous Bow</td>
<td>(4.857, 4.857)</td>
<td>4.857</td>
<td>0.0</td>
</tr>
<tr>
<td>D&lt;sub&gt;bulb&lt;/sub&gt;</td>
<td>Draft of Bulbous Bow</td>
<td>(4.389, 4.389)</td>
<td>4.389</td>
<td>0.0</td>
</tr>
<tr>
<td>L&lt;sub&gt;bulb&lt;/sub&gt;</td>
<td>Length of Bulbous Bow</td>
<td>(4.974, 4.976)</td>
<td>4.975</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 12.4: Hull Design Randomly Generated Hull Form parameters for lines shown in figure 12.32. Also, rendered surface and curvature shown in figure 12.33.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Form Parameter</th>
<th>Target Value (1)</th>
<th>Actual Value (2)</th>
<th>% Δ error</th>
</tr>
</thead>
<tbody>
<tr>
<td>∇</td>
<td>Area of SAC curve, (Volume of Ship)</td>
<td>ia(30821.066,30821.07)</td>
<td>30821.068</td>
<td>0.0</td>
</tr>
<tr>
<td>L_WL</td>
<td>Waterline Length</td>
<td>ia(118.211,118.217)</td>
<td>118.214</td>
<td>0.0</td>
</tr>
<tr>
<td>B</td>
<td>Ship Beam</td>
<td>ia(28.143,28.146)</td>
<td>28.145</td>
<td>0.0</td>
</tr>
<tr>
<td>D</td>
<td>Ship Draft</td>
<td>ia(15.102,15.104)</td>
<td>15.103</td>
<td>0.0</td>
</tr>
<tr>
<td>A_M</td>
<td>Midship Section Area</td>
<td>ia(382.554,382.56)</td>
<td>382.557</td>
<td>0.0</td>
</tr>
<tr>
<td>A_Cp</td>
<td>Center-plane Area</td>
<td>ia(1562.206,1562.212)</td>
<td>1562.209</td>
<td>0.0</td>
</tr>
<tr>
<td>A_WP</td>
<td>Waterline Area</td>
<td>ia(3156.981,3156.987)</td>
<td>3156.984</td>
<td>0.0</td>
</tr>
<tr>
<td>LCB</td>
<td>Longitudinal Center of Buoyancy</td>
<td>ia(60.494,60.5)</td>
<td>60.497</td>
<td>0.0</td>
</tr>
<tr>
<td>C_B</td>
<td>Block Coefficient</td>
<td>ia(0.613,0.613)</td>
<td>0.613</td>
<td>-0.0</td>
</tr>
<tr>
<td>C_P</td>
<td>Prismatic Coefficient</td>
<td>ia(0.681,0.682)</td>
<td>0.682</td>
<td>0.006</td>
</tr>
<tr>
<td>C_M</td>
<td>Midship Area coefficient</td>
<td>ia(0.9,0.9)</td>
<td>0.9</td>
<td>-0.006</td>
</tr>
<tr>
<td>CWL</td>
<td>Waterline area coefficient</td>
<td>ia(0.949,0.949)</td>
<td>0.949</td>
<td>-0.0</td>
</tr>
<tr>
<td>L_FWL</td>
<td>Length of Flat of Waterline</td>
<td>ia(12.472,12.478)</td>
<td>12.475</td>
<td>0.0</td>
</tr>
<tr>
<td>C_PK</td>
<td>Center-plane Keel Profile area coeeficient</td>
<td>ia(0.875,0.875)</td>
<td>0.875</td>
<td>-0.0</td>
</tr>
<tr>
<td>L_FCpk</td>
<td>Length of the Center-plane Keel Profile</td>
<td>ia(9.354,9.36)</td>
<td>9.357</td>
<td>-0.0</td>
</tr>
<tr>
<td>L_FSAC</td>
<td>Length of Flat of SAC</td>
<td>ia(8.817,8.823)</td>
<td>8.82</td>
<td>0.0</td>
</tr>
<tr>
<td>DLR</td>
<td>Displacement to Length Ratio</td>
<td>ia(0.019,0.019)</td>
<td>0.019</td>
<td>-0.0</td>
</tr>
<tr>
<td>∇Bulb</td>
<td>Area of Bulbous Bow</td>
<td>ia(60.811,60.811)</td>
<td>60.811</td>
<td>0.0</td>
</tr>
<tr>
<td>CAXBulb</td>
<td>bulb sectional area maximum</td>
<td>ia(0.732,0.732)</td>
<td>0.732</td>
<td>-0.0</td>
</tr>
<tr>
<td>CAWPbulb</td>
<td>Water-plane Bulb Coefficient</td>
<td>ia(0.72,0.72)</td>
<td>0.72</td>
<td>-0.0</td>
</tr>
<tr>
<td>CCPBulb</td>
<td>Center-plane Bulb Coefficient</td>
<td>ia(0.867,0.867)</td>
<td>0.867</td>
<td>-0.0</td>
</tr>
<tr>
<td>CBbulb</td>
<td>Bulb Block Coefficient</td>
<td>ia(0.732,0.732)</td>
<td>0.732</td>
<td>-0.0</td>
</tr>
<tr>
<td>CPhbulb</td>
<td>Bulb Prismatic Coefficient</td>
<td>ia(1.0,1.0)</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CB</td>
<td>Breadth Ratio</td>
<td>ia(0.16,0.161)</td>
<td>0.16</td>
<td>-0.0</td>
</tr>
<tr>
<td>C_plr</td>
<td>Ship Length to bulb length ratio</td>
<td>ia(0.036,0.036)</td>
<td>0.036</td>
<td>-0.0</td>
</tr>
<tr>
<td>C_zb</td>
<td>Ship Draft to Bulb height Ratio</td>
<td>ia(0.286,0.286)</td>
<td>0.286</td>
<td>-0.0</td>
</tr>
<tr>
<td>C_abt</td>
<td>Midship to Mid bulb area ratio</td>
<td>ia(0.037,0.037)</td>
<td>0.037</td>
<td>-0.0</td>
</tr>
<tr>
<td>C_abl</td>
<td>Center-plane Ratio</td>
<td>ia(0.01,0.01)</td>
<td>0.01</td>
<td>-0.0</td>
</tr>
<tr>
<td>CVPR</td>
<td>Displacement Ratio</td>
<td>ia(0.002,0.002)</td>
<td>0.002</td>
<td>0.0</td>
</tr>
<tr>
<td>B_bulb</td>
<td>Beam of Bulbous Bow</td>
<td>ia(4.517,4.517)</td>
<td>4.517</td>
<td>0.0</td>
</tr>
<tr>
<td>D_bulb</td>
<td>Draft of Bulbous Bow</td>
<td>ia(4.32,4.32)</td>
<td>4.32</td>
<td>0.0</td>
</tr>
<tr>
<td>L_bulb</td>
<td>Length of Bulbous Bow</td>
<td>ia(4.26,4.26)</td>
<td>4.26</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 12.5: Hull Design Randomly Generated Hull Form parameters; center profile keel curve integrated with Bulbous bow
Chapter 13

Summary and Conclusions

As shown in the tables 12.1, 12.1, 12.3, and 12.4, the difference between incoming form parameter design vectors and the resulting parameters output from the curves directly generated by form parameter design is basically controlled by the desired accuracy of the solvers. There is no need to allow for “wiggle room” whereby inconsistent equality constraints would be relaxed into inequalities so that valid solutions to the optimization problems could be found.

Though in practice this relaxation may of course be preferred, simply for lack of preference on certain design variables, what we have demonstrated here is the inherent capability to decide feasibility beforehand — as exposted by the fact that the constraint system eliminates infeasible choices from the design space. Furthermore we have shown the capability to generate systems of consistent equality constraints on a scale relevant to the engineering design of ship hull form geometry.

A feasible form parameter system has been constructed and shown to work by constructing bare hulls and bulbous bows which meet kraft parameters and hull form parameters.

The system generates complex hull forms using the truncated hierarchical B-spline (THB-spline) methodology to combine surfaces in a manner which guarantees feasibility with form parameters. The THB-splines add additional freedom to control the boundary conditions at surface connections, or otherwise add local detail to the geometry.

Novel Contributions

Original work is found in applying interval arithmetic, automatic differentiation, and relational programming methodologies to the problem of feasible ship hull design by form parameter design. Additionally, the B-splines are extended to more adaptable THB-splines, and these are used to construct an adaptive curve
solver and compatible surface definition.

The primary new features added are the capability to reason from a hull design specification to a consistent set of form parameters with success in an automated setting, even for cases of random design parameter selection, to automatically infer when a constraint set is infeasible, to speed the process of form parameter design by narrowing the search space to the bounds of the feasible domain, and to provide an adaptive solver capable of complex geometry with local shape and boundary interface control.

Issues, Limitations, and Lessons Learned

Issues and limitations are listed below. Best practices and corrections to address the limitations are listed as well.

- As might be guessed, the methodology is not infallible.
  - It is only as correct as the rules are able to make it. Curve solutions still fail to produce usable geometry due to the nonlinear nature of the curve optimization problem and the many constraints.
  - Interval methods cannot always reduce a system to its true feasible domain immediately due to the interval dependency problem. Therefore, it will always be possible for the form parameter search tool to pick infeasible form parameters, though in practice, this seems rare for a well tuned system. Further, when an infeasible form parameter is chosen, this rapidly becomes apparent, since the design space collapses to the null set. Backtracking then ensures the mistake is erased and a feasible form parameter is substituted.

- We have now moved the problem of specifying feasible form parameters to that of specifying feasible design spaces. Hopefully, this is an easier problem. In practice though, users may still desire a particular ship design. If this design is in some way infeasible, the user should get warning of this fact from the tool. This may, nevertheless, prove frustrating for some.

- Constraint solvers have some native weaknesses, especially to do with solver performance and efficiency: There is some sensitivity to the order of search.

  In practice often the best ordering can be heuristically guided by the engineer, as the “hardest” constraints should be tackled first.
For instance, a given set of bulb form parameter rules was developed over time and interfaced with the bare hull form solver. At that point, the bare hull solver began specifying length and cross sectional area form parameters for the bulb. But the bulb design space search had been developed to search over other coefficients first, as heuristically, it was found that often narrowing more global parameters first was beneficial to search efficiency. This degraded search efficiency when the first inputs were actually area and length, and came from the bare hull solver. Thus a new heuristic for rules search might be to pick form parameters from the most tightly constrained, or first defined, variables first.

In this case, the bulbous bow design search was improved by switching from other coefficients, to length and cross section area, since more design space data (constraints) existed for these parameters. The interval literature has much more to say about picking “which box to prune next” — which is closely related problem. See Kearfott [103] for details.

Finally, in developing the software here, the function `run_binary_relation` actually runs a three term relational rule four times, initially computing with every term reified to its actual value in the environment, and the three more times, one each with a particular variable left in logical form. It was found that this induced the system to more fully compute the propagations across the three terms. There is likely a more efficient way to handle this computation but that waits for future development time.

**Future Work and Outlook**

**Feature Complete Feasible FPD**

The solver here is in many ways rudimentary, compared to a full featured form parameter design program. In the future, additional curves of form should be incorporated to specify such quantities as, for example, transverse curve end conditions. These are specified by curves of tangential and curvature form along the length of the hull. Such curves of form could alter the formulation of the hull design problem used in this work. We would no longer need to impose as many boundary conditions on the hull “from without.”
Constraint Solver Performance

Performance of the constraint solver could be increased in many ways. Term re-writers [107], monotonic methods [169], and more sophisticated interval bounding methods [215] could all be tried in the future. Since we do not attempt to rigorously solve the global hull design problem for “optimality” (whatever this could mean in such context), outward rounding could be neglected for perhaps some speedup. More utility might be derived from introducing alternate contractors for local and even global consistencies, for instance the interval Newton’s method. Of course, the traditional rules of program optimization apply. Our computations were performed on a standard Python and numpy installation, without the benefit of BLAS or LAPACK. A new version might take advantage of more efficient libraries or language implementations.

Geometric Accuracy

Form parameter design quality is sensitive to certain hyperparameters, such as the placement of transverse curve stations. Furthermore, very well conditioned form parameter design systems cannot be perfect, since the match between final geometry and specified form parameters is only implicit in the formulation, and not an explicit result of an particular solver.

While it is true that the curves which are so optimized will meet their form parameters to within a specified tolerance, that does not necessarily mean that the resulting surface geometry will meet the same form parameters with the same tolerance. See for example the masters thesis of Sanches [216] for results and comparisons with target designs using professional form parameter software. Form parameter design gets relatively close, but it is not the perfection of an end to end solved solution.

Another method of ensuring accurate surface representation of the form parameter inputs would be to do variational form parameter optimization and constraint solving of the surfaces directly. One complicating factor here is the expense of computing derivatives of complicated surface expressions with many free variables of design. Fortunately, it is known from experimentation with the AD tool presented in this thesis, that it is possible to compute gradients and Hessians of B-spline surfaces and B-spline surface quantities, including average and Gaussian curvatures, via automatic differentiation. Performance of such a solver has not been studied, but some of the traditional means, i.e. compiling to Fortran or C++, might need to be combined with the AD tool. Of potential interest here, the Fortran standard is now sophisticated enough to readily and simply support various methods of automatic differentiation [217] and note that prior versions of
Fortran have been used in an interval optimization context [103]. Alternatively, there are many methods of speeding up Python or outsourcing computations to the GPU. Perhaps with improvements in the algorithms or porting to faster languages, direct surface optimization will be shown to be feasible and superior to the curve based methods given here.

**Hyperparameter Search**

As mentioned above, form parameter design of ship hull forms comes with certain hyperparameters, such as the stations at which transverse section curves will be computed, and which curves of form will be utilized in supplying the form parameters with their various constraint inputs.

Optimization and machine learning literature has shown many methods of hyperparameter tuning. Some methods are,

- Bayesian Optimization
- Meta-modeling, surrogate modeling, or response surface modeling or hyperparameters

Some of these techniques might be used in furthering the state of the art of ship hull form parameter design.

Note that machine learning for hyperparameter optimization is different from machine learning to build an efficient approximator of the design space itself. Hyperparameter optimization is a much more limited problem, though of course related, since good hyperparameters aid in allowing the algorithms to find good solutions to the overall problem.

**Complementarity with Machine Learning Approaches**

The system given in this dissertation is envisioned as underpinning a more complex ship design system which would use the methodology of this thesis to generate hull geometry. This geometry would be further processed by analytic programs for evaluating wave resistance, ship motions, etc. Only when these additional analysis have been carried out, can the hull form performance finally be assessed realistically. When developing a general engineering hull design analysis and optimization system on top of the system described in this paper, the rules system would perhaps serve as a “reasonable prior” for more sophisticated machine learning approaches, such as reinforcement learning [218]. That is, the system here would be the reference point which would facilitate the training of superior systems for the same job. Such systems would leverage analytic tools to judge and optimize hull form performance.
The constraint system given here could be used to establish some baseline of what feasible designs are. This will aid, for example, a reinforcement learning algorithm in learning to generate useful information about the hull design space. The idea is to have a simple and robust generative system so that the system can learn from a baseline of designs that are not so bad that the algorithm spends a great deal of time learning very basic design characteristics. The immediate need, in creating a reinforcement system to learn from the current methods, is some kind of scoring system for the geometry which is generated. Likely we can rely on the extensive literature of hull optimization by CFD and other tools, to find the right methodology here. This will be problem dependent and so we need say no more at this time. More generally, constraint logic programming systems can aid machine learning approaches in the following ways:

- Machine learning is data hungry, where constraint programming is data generative.
- Machine learning is opaque, while constraint programming is logical and human understandable.

Additional areas where both machine learning and constraint programming are weak are:

- Both machine learning and constraint programming are narrow, but constraint programming can be generalized by writing down more general constraints, or constraints for other systems. This might be used to help generate machine learning systems that could be tailored quickly for one problem, and then easily switched to another, if the machine learning system is generated via constraint and declarative methods.

- Both machine learning and constraint programming are brittle, but they are brittle in different ways. In this thesis we have seen that constraint programming is quite robust to random inputs within a design space, but one could easily break the system using infeasible rules. On the other hand, machine learning require large amounts of data to generalize to new inputs, and are still easily broken by data. Here, the advantageous use of machine learning and constraint programming is perhaps less clear.

In the end, machine learning, at its most basic, is a type of optimization procedure. The complementarity between optimization and constraint programming has been remarked upon before [124]. It might be conjectured that if machine learning approaches begin, once more, to hit diminishing marginal returns, the constraint programming might represent another wave of artificial intelligence progress, waiting in the wings. Indeed, an analysis of trends of computing, towards ever more abstract systems, and a need for tighter validation and control over large swaths of code and data, constraint programming might just be the paradigm
to watch, as supposed by Wallace in [122]. If so, then interval constraint programming would be the paradigm to follow within numerical analysis and engineering applications.

**Automatically Generating Custom Curve Solvers**

A large effort to generate custom curve solvers by hand is currently necessary when generating form parameter design systems. This needs to be reduced. One can think of using the automated rules system, together with the composable automatic differentiation system, to automatically compose functionals that are a good fit to the particular design problems that come up when designing, e.g. hull curves, section curves, and detail curves, and also across different design types, such as offshore supply vessels, destroyers, container ships, cruisers, etc.

**Geometric Transformations of Sub-problems — Developable Surfaces**

Recent developments in computational geometry concern the use of exterior differential geometric methods in the discrete setting [219]. Some of these methods might be exploited to, for instance, attack the problem of developable design of the ship hull form while avoiding the most expensive methods of minimizing Gaussian curvature, for instance by using techniques derived from such papers as Stein [220]. These methods could be investigated for use in form parameter design in complement to research from within naval architecture, typically using rulings between space curves and modifications to this technique in support of more robust generation of developable surfaces [221, 222]

Of particular interest to this author are the ideas of combining AD driven surface optimization with the clever mathematical transformations seen in the computational mesh and manifold design techniques from [219]. This is mere conjecture, but perhaps those algorithms and their methods of reducing complicated nonlinear optimization problems to linear time algorithms [223] could help to enable more efficient form parameter design of surfaces. This could enable exact fits between form parameter design inputs and resultant geometry across all constraints, and do so at potentially reasonable computational cost. This would complete the robustness and accuracy of the form parameter system, with, on the input side, constraint logic to pick consistent sets of form parameter input vectors, and, on the output side, exterior differential algorithms for efficient, numerically exact, and robust solutions of the desired geometry itself.
Scalability — Cloud Computing Application

The work presented here is fundamentally well adapted for driving parallel design workloads. Some changes to the search pattern would be advisable in order to make best use of the increased ability to search.

- For breadth first search, simply start multiple search processes with differing form parameters.
  - One could generate a Sobol sequence of starting parameters across form parameters. Other methods of partitioning are also available in the meta-modeling literature.
  - More sophisticated methods should employ meta-learning of the design space with further measures of merit for the resulting geometry.

- An alternative might be that for each time the search algorithm splits the space, allocate a separate process to each partition.


URL http://dx.doi.org/10.1016/j.cad.2006.02.005

[31] J. Peters, A. Nasri, Computing volumes of solids enclosed by recursive subdivision surfaces 16 (3)


Research Center (1987).

[34] D. R. Forsey, R. H. Bartels, Hierarchical b-spline refinement, in: Proceedings of the 15th Annual
Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1988, Atlanta, Georgia,
URL http://doi.acm.org/10.1145/54852.378512


[36] A.-V. Vuong, C. Giannelli, B. Jüttler, B. Simeon, A hierarchical approach to adaptive local refinement
in isogeometric analysis, Computer Methods in Applied Mechanics and Engineering 200 (49) (2011)
S0045782511002933

[37] P. Bornemann, F. Cirak, A subdivision-based implementation of the hierarchical b-spline finite
doi:https://doi.org/10.1016/j.cma.2012.06.023.
S0045782512002204

[38] T. W. Sederberg, J. Zheng, A. Bakenov, A. Nasri, T-splines and t-nurccs, ACM Trans. Graph. 22 (3)
URL http://doi.acm.org/10.1145/882262.882295

URL http://doi.acm.org/10.1145/1015706.1015715

[40] T. Dokken, T. Lyche, K. F. Pettersen, Polynomial splines over locally refined box-
S0167839613000113

[41] K. A. Johannessen, F. Remonato, T. Kvamsdal, On the similarities and differences between classical
hierarchical, truncated hierarchical and lr b-splines, Computer Methods in Applied Mechanics and En-
S0045782515001073


URL http://dx.doi.org/10.1007/s10444-013-9315-2


[50] E. Evans, M. Scott, X. Li, D. Thomas, Hierarchical t-splines: Analysis-suitability, bézier extraction, and application as an adaptive basis for isogeometric analysis 284.


[53] T. Sederberg, S. Parry, Free-form deformation of solid geometric models, SIGGRAPH.


[69] C. Mizine, I. Rogers, B. Wintersteen, Hull form variation in the early stage design synthesis process, Naval Engineers Journal.
[70] W. Wilson, D. Hendrix, J. Gorski, Hull form optimization for early stage ship design, Naval Engineers Journal 122.


[84] M. Parsons, Applications of optimization in early stage ship design, Ship Science and Technology 3.


266


URL http://arxiv.org/abs/1605.02688


267


271


[176] W. M. Kahan, A more complete arithmetic, lecture notes for a summer course at the university of michigan (1968).


URL https://doi.org/10.1007/s10817-016-9397-x

URL http://dx.doi.org/10.1007/978-3-642-12251-4_19


URL https://doi.org/10.1023/A:1026485406803

URL http://ieeexplore.ieee.org/servlet/opac?punumber=7140719

URL http://doi.acm.org/10.1145/321250.321253


[200] X. Wei, Y. Zhang, Y. J. Zhanga, T. J. R. Hughes, Volumetric truncated hierarchical spline construction on unstructured hexahedral meshes for isogeometric analysis applications.


[202] Q. Li, X. Ma, R. Cripps, P. Lin, A subdivision-based implementation of non-uniform local refinement with thb-splines.
URL https://hull-repository.worktribe.com/output/436845


URL http://dx.doi.org/10.1016/j.cagd.2007.12.005


URL https://doi.org/10.1007/978-3-642-54382-1_18

URL https://doi.org/10.1007/s00211-015-0711-z


URL https://doi.org/10.1007/s11831-017-9220-9


URL http://dx.doi.org/10.1007/978-3-7091-6918-6_14


URL http://doi.acm.org/10.1145/2504435.2504442
The author is a practicing software developer in the fields of science and engineering. Born and raised in Alabama, he explored aerospace engineering and economics before moving to New Orleans to pursue naval architecture and computational software development. In his spare time he enjoys studying physics and differential geometry.