

# **Direct unlofting for geometric design (Report From Team 4)**

**Mentor: Thomas A. Grandine (Boeing)**

Yilin Dai, Michigan Technological University  
Christina Dekany, Southern Methodist University  
Simon Gemmrich, McGill University  
Jiyung Lois Kwon, Washington State University  
Zhongyi Nie, University of Kentucky  
Qiling Shi, University of Central Florida

**Mathematical Modeling in Industry XII  
Institute for Mathematics and its Applications (IMA)  
August 6-15, 2008**

# 1 Introduction

The Boeing Company makes intensive use of computer automated shape design tools for airplanes. One of their geometric tools is a proprietary *Python*-based software package called *Geoduck*. In the context of shape design, the process of creating the outer mold line of an aircraft from a given set of dimensional engineering and physical measurements and shape requirements is called lofting. The present report examines the inverse process, which is called the unlofting procedure. This inverse procedure is closely tied to the direct problem. In practice it is usually used to align geometric shapes created by different lofting methods. The unlofted parameters can then be used to further optimize the given shape.

Thomas A. Grandine from Boeing posed an unlofting problem at the 12th Mathematical Modeling in Industry Workshop for graduate students, held at the IMA on August 6-15, 2008. Given a simplified lofting procedure written in *Geoduck* and a baseline configuration of an airplane designed by engineers, the challenge is to find the optimal lofting parameters using the lofting procedure with respect to the baseline configuration. For any lofted airplane we measure its 'distance' from the baseline configuration and formulate the lofting parameters accordingly. This is done by sampling points on the baseline configuration, projecting them onto the mathematically modeled plane, and measuring the residual vector of the distances between the sampled and projected points. This leads to the least-squares formulation of the unlofting problem. However, among other complications, the highly nonlinear nature of the lofting procedure as well as the nonlinearity and nondifferentiability of the point-projection method make this nonlinear least-squares problem difficult to solve.

To devise an unlofting procedure, we look at three examples: arc-matching, a simple airplane design, and a harder airplane design. The internal nonlinear least-squares solver of *Geoduck* is an implementation of the Levenberg-Marquardt algorithm and we use it extensively for the first part of our assignment. For the second step we implement a more flexible solver in *MATLAB*, which allows us to adjust its behavior to the specific problem. Zack Thunemann from Boeing helped us building up the connection between *MATLAB* and *Geoduck*. We use this to take advantage of the nonlinear least-squares tools in *MATLAB*.

## 2 The nonlinear least-squares problem

Assume that we are given a geometric lofting procedure  $\mathbf{G}$  that takes a vector  $\boldsymbol{\nu} \in \mathcal{H} \subseteq \mathbb{R}^M$  of  $M$  physical design parameters and outputs the geometric shape  $\mathbf{G}(\boldsymbol{\nu})$  of an airplane. The feasible set of parameters  $\mathcal{H}$  is a known hypercube in  $\mathbb{R}^M$ . Now, for a given baseline configuration of an airplane shape we sample  $N$  points,  $\mathbf{P0}_i$ , as a preprocessing step and project these onto the plane given by  $\mathbf{G}(\boldsymbol{\nu})$  to get the points  $\mathbf{P1}_i$ . Let the function  $\mathbf{f}(\mathbf{x})$  take the vector of design parameters and output the residual vector of the distances between  $\mathbf{P0}_i$  and  $\mathbf{P1}_i$ ,  $i = 1, \dots, N$ . We are interested in solving the following least-squares problem:

$$\min_{\boldsymbol{\nu} \in \mathcal{H}} \sum_{i=1}^N (\text{dist}(\mathbf{P0}_i, \mathbf{P1}_i))^2 \quad (1)$$

### 2.1 The Secant Levenberg-Marquardt Method

There are few things we need to consider when we choose an appropriate algorithm to fit in our specific problem. The first one is that the function in our problem is not differentiable due to the configuration of the airplane surfaces. This means that we cannot use the derivatives to produce appropriate Jacobian or Hessian matrices that are used in most nonlinear least squares solvers. Instead of computing the derivatives, we use finite differences methods to obtain the numerical approximations of the needed derivatives. The first solver we use is a secant version of the Levenberg-Marquardt Method. The Levenberg-Marquardt Method is based on the Gauss-Newton Method.

#### Algorithm Secant Version of the L – M Method

```
k = 0; x = x0; B = B0; j = 0
G = BTf(x); found = (||g||∞ ≤ ε1)
while (not found) and (k < kmax)
k = k + 1; Solve(BTB + μI)h = -g
if ||h|| ≤ ε2(||x||) + ε2
    found = true
else
    for i = 1 : m
    j = mod(j, n) + 1; if ||hj|| < 0.8||h||, update B using xnew = x + ηej
    xnew = x + h; Update B using xnew
```

```

    if  $F(\mathbf{x}_{\text{new}}) < F(\mathbf{x})$ 
       $\mathbf{x} = \mathbf{x}_{\text{new}}$ 
       $\mathbf{g} = \mathbf{B}^T \mathbf{f}(\mathbf{x}); \text{found} = (\|\mathbf{g}\|_\infty \leq \epsilon_1)$ 
    end

```

The L-M Method introduces a damping parameter,  $\mu > 0$ , which influences both the direction and the size of the guess leading to the next step. During each iteration,  $\mu$  guarantees that  $\mathbf{h}$ , the current step, is on the descent direction by making the coefficient matrix positive definite:

$$(\mathbf{B}^T \mathbf{B} + \mu \mathbf{I}) \mathbf{H} = \mathbf{B}^T \mathbf{f}(\mathbf{x})$$

Also, it will adjust the step size so that we do not need a specific line search. A large  $\mu$  generates better iterations if the current  $\mathbf{x}$  is far from the final approximation  $\mathbf{x}^*$ , and a small  $\mu$  gives fast convergence for  $\mathbf{x}$  close to  $\mathbf{x}^*$ . In addition, during the iterations the finite difference approximation matrix  $\mathbf{B}$  of Jacobian matrix  $\mathbf{J}$  (in the Secant version of L-M Method) is updated to find a better direction to the minimizer.

The L-M method has a condition such that the set of all steps ( $\mathbf{h}_k = \mathbf{x}_k - \mathbf{x}_{k-1}$ ) spans the whole of  $\mathbf{R}^n$ . For each  $\mathbf{h}$ , the method makes sure that the  $\mathbf{h}$  has a different direction from any previous  $\mathbf{h}$ 's. This  $\mathbf{x}$  updates the matrix  $\mathbf{B}$ . Based on these characteristics the Secant Version of the Levenberg-Marquardt Method satisfies our non-differentiable nonlinear least-squares problem. One of the problems, however, with this algorithm is that because it uses the finite difference approximation instead of Jacobi, it needs additional conditions to guarantee that the next iteration will move toward the minimizer. This requires updating the finite difference approximation many times during one iteration, which means it will frequently call the function  $\mathbf{f}(\mathbf{x})$  in *Geoduck*. As a result, this algorithm may take few iterations to get close to the minimizer, but it does not improve the efficiency. Furthermore, in practice, it does not have a good performance since *Geoduck* needs a lot of time to generate  $\mathbf{f}(\mathbf{x})$ . We try to find another algorithm which will not call  $\mathbf{f}(\mathbf{x})$  as many times during one iteration. This leads to the following BFGS algorithm.

## 2.2 The BFGS Method

This method is based on the The Quasi-Newton Method which will update Hessian matrix once during each iteration and therefore, in our case, it will

call  $f(\mathbf{x})$  in *Geoduck* fewer times. In this specific problem we hope it will improve the efficiency of our procedure.

Input variables:  $n \in \mathbf{Z}$ ,  $\mathbf{x}^c \in \mathbb{R}^n$ ,  $\mathbf{x}^+ \in \mathbb{R}^n$ ,  $\mathbf{g}^c \in \mathbb{R}^n$ ,  $\mathbf{g}^+ \in \mathbb{R}^n$ ,  $\varepsilon \in \mathbb{R}$ ,  $\eta \in \mathbb{R}$ , *angrad*  $\in$  Boolean (TRUE if analytic gradient is used, FALSE otherwise).

Input-output variables:  $\mathbf{H} \in \mathbb{R}^{n \times n}$  symmetric

Output variables: none

#### Algorithm

1.  $\mathbf{s} = \mathbf{x}^+ - \mathbf{x}^c$
2.  $\mathbf{y} = \mathbf{g}^+ - \mathbf{g}^c$
3.  $\text{temp}_1 = \mathbf{y}^T \mathbf{s}$
4. IF  $\text{temp}_1 \geq \varepsilon^{1/2} \|\mathbf{s}\|_2 \|\mathbf{y}\|_2$ , THEN  
(ELSE update is skipped and algorithm terminates)
  - (a) IF *angrad* = TRUE, THEN  $\text{tol} = \eta$   
ELSE  $\text{tol} = \eta^{1/2}$
  - (b) skipupdate  $\leftarrow$  TRUE
  - (c) FOR  $i = 1$  TO  $n$  DO
    - i.  $t_i \leftarrow \sum_{j=1}^i \mathbf{H}_{j,i} \mathbf{s}_j + \sum_{j=i+1}^n \mathbf{H}_{i,j} \mathbf{s}_j$
    - ii. IF  $|y_i - t_i| \geq \text{tol} \max\{|g_i^c|, |g_i^+|\}$ , THEN: skipupdate = FALSE.
  - (d) IF skipupdate = FALSE, THEN
    - i.  $\text{temp}_2 = \mathbf{s}^T \mathbf{t}$
    - ii. FOR  $i = 1$  TO  $n$  DO, FOR  $j = i$  TO  $n$  DO:  
 $\mathbf{H}_{i,j} = \mathbf{H}_{i,j} + (\mathbf{y}_i \mathbf{y}_j) / (\text{temp}_1) - (\mathbf{t}_i \mathbf{t}_j) / (\text{temp}_2)$

The main idea of the Quasi-Newton Method is to use an approximation of the inverse Hessian matrix,  $\mathbf{H}^{-1} = \mathbf{B}$ . Since computing of the inverse Hessian matrix directly requires many computations and is inefficient, alternative ways of finding the inverse of Hessian matrix  $\mathbf{H}$  are used in optimization problems. Numerically approximating the  $\mathbf{H}^{-1}$  in the BFGS method

is known as the best performing method. The BFGS method updates the inverse of the Hessian matrix  $\mathbf{H}^{-1}$  by the formula

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \left( \frac{1 + \mathbf{q}_k^T \mathbf{B}_k \mathbf{q}_k}{\mathbf{q}_k^T \mathbf{p}_k} \right) \frac{\mathbf{p}_k \mathbf{p}_k^T}{\mathbf{p}_k^T \mathbf{q}_k} - \frac{\mathbf{p}_k \mathbf{q}_k^T \mathbf{B}_k + \mathbf{B}_k \mathbf{q}_k \mathbf{p}_k^T}{\mathbf{q}_k^T \mathbf{p}_k} \quad (2)$$

where  $\mathbf{B}_k = \mathbf{H}_k^{-1}$ ,  $\mathbf{q}_k = \mathbf{H} \mathbf{p}_k$ , and  $\mathbf{p}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ . The matrix  $\mathbf{B}$  is computed only once per iteration so the approximation is rough if the number of parameters is small, but it has a superlinear convergence rate.

## 3 Method Introduction

### 3.1 Sampling Strategy

First we need to develop a method to measure the difference between those two planes so that we can tell whether they match each other or not. We do this by sampling points on the engineer’s plane, projecting them onto our plane, and then computation the distance between the sampling points and their projections.

Our first sampling method involves picking equidistant points along the grid of the unit square preimage of the plane and then mapping these selected points onto the corresponding surfaces of the engineer’s plane. For example, for each piece of the plane, such as the body loft, we create a list of sampling points. Future work will go into a sampling method that takes advantage of the curves of the surfaces and samples according to the shape of the piece, rather than a strict equidistant sampling.

### 3.2 The Objective Function $f(\mathbf{x})$ in *Geoduck*

The objective function used in the nonlinear solver takes as input the vector of parameter value guesses and returns a residual vector. Within the method, that is, we create a geometry generator plane using the input vector and project our sampled points from the engineer’s plane onto this new plane. The residual vector is the numerical value of the distance between the sampled point and the projected point. This is the vector that the nonlinear solver is to minimize.

### 3.3 Rescaling strategies and penalty functions

Within the objective function we must check the physical feasibility of the nonlinear solver’s guess vector. The nonlinear solver is independent from the plane parameters and therefore many times will leave the physical boundaries that limit the design of a plane.

We have two separate approaches for ensuring that the nonlinear solver computes a list of parameters that have a physical correlation. Our first method involves placing a check within the objective function that analyzes the nonlinear solver’s guess vector. If the function determines that a particular parameter value is out of the physical region, it will reset this value to a guess in the middle of the upper and lower bounds of this region. Our results in the following sections show that though this method is logically not the best mathematical implementation for the solver, it does allow our method to produce desirable results. One of the problems with this check function is that the nonlinear solver can at times contradict itself and be lead astray. It may be on a path where each time it guesses a vector, the check function resets it so that it must go down that path again. When this happens, we use our artistic mathematical abilities and give the vector of parameter values a nudge in a direction that will hopefully allow the nonlinear solver to converge to the correct answer. Unfortunately, this approach introduces unwanted discontinuous behavior to the objective function and should be avoided.

One possible remedy is the use of penalty methods. They are commonly used to solve constraint optimization problems. The basic idea is to penalize parameter values outside of a certain region. In the case of the optimization problem in equation (1), we exchange the hypercube constraint with additional terms on the right hand side, i.e. we try to solve the following minimization problem instead:

$$\min_{\boldsymbol{\nu} \in \mathbb{R}^M} \sum_{i=1}^N (\text{dist}(\mathbf{P}_i, \mathbf{G}(\boldsymbol{\nu})))^2 + \sum_{j=1}^M \mathbf{g}_j^2(\boldsymbol{\nu}_j). \quad (3)$$

Here, we chose the penalty term  $\mathbf{g}_j(\boldsymbol{\nu}_j)$  such that it vanishes if the parameter  $\boldsymbol{\nu}_j$  is contained in the interval  $\mathbf{I}_j = [\mathbf{a}_j, \mathbf{b}_j]$  and increases logarithmically with growing distance  $\text{dist}(\boldsymbol{\nu}_j, \mathbf{I}_j)$ . Heuristically, these penalty terms should then automatically drive the parameters into the feasible region.

### 3.4 The interface between *MATLAB* and *Geoduck*

*Geoduck* provides only a single nonlinear least-squares solver. It is a rather unstable implementation of the Levenberg-Marquardt algorithm written for unconstrained optimization problems. Given that we had no access to its source code, we had to treat it as a black box solver. An interface between *Geoduck* and *MATLAB* was provided to us some time into the project and greatly increased the possible choices of solvers. We use a client-server relation between the programs so that *Geoduck* acts as a server for the client, *MATLAB*, and returns the residual vector when passed a vector of parameter values. *MATLAB* has several nonlinear least-squares methods implemented and we wrote our own computer code adapted to the specific requirements for the following two methods:

- LMDIF, a Levenberg-Marquardt method which uses a finite difference approximation for the Jacobian and
- BFGS, (Broyden-Fletcher-Goldfarb-Shanno) which is a Quasi-Newton method.

For all those methods we need to call *Geoduck*'s geometry generator and the projection routines from within *MATLAB*. This client-server relationship is based solely on the exchange of string variables. Hence, a *MATLAB* function is needed to transform a *Geoduck* command into a string to pass it. The command is then executed in *Geoduck*. Similarly, the computed *Geoduck* data vectors need to be converted to string variables element by element and transferred back.

## 4 First step: Unlofting an arc

We test our implemented secant version of the L-M code with an arc-matching problem. The arc-matching problem is that we have an arc generated by the geometry generator. Using our *MATLAB* version of the secant L-M method, we match the parameters of an initial line onto the arc.

Computing  $f(x)$  in *Geoduck* requires considerable computing power since the iterative nonlinear solver usually evaluates the objective function at many different trial points. In order to avoid extra complications, we therefore test our program on the problem of unlofting a single arc. The main idea and procedure are the same as in the airplane unlofting problem we

are interested in. Given an arc characterized by two parameters we consider sample nine points. We send our starting guess to the objective function, which will generate an arc based on this guess, project the nine sampled points onto the generated arc, and return the distance between the sampled points and their projections.

We can see in figures 1, 2, and 3 that the performance of our implemented versions of the Secant L-M method (LMDIF) and the BFGS method are satisfactory.

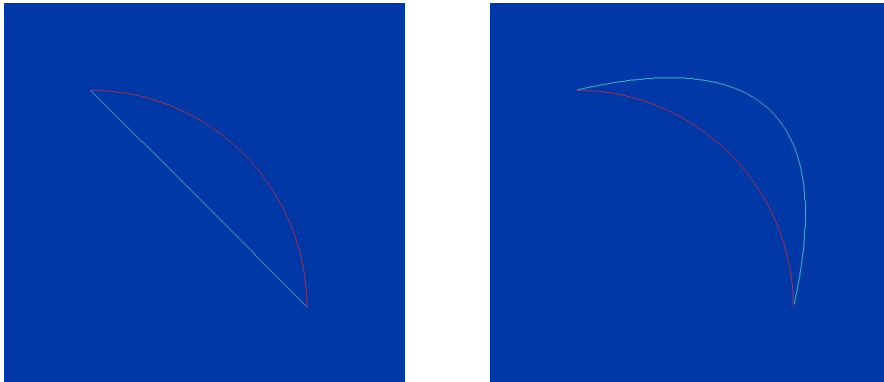


Figure 1: The CAD designed arc and the Geometry Generated arc (after 0 BFGS iteration and 2 BFGS iterations)

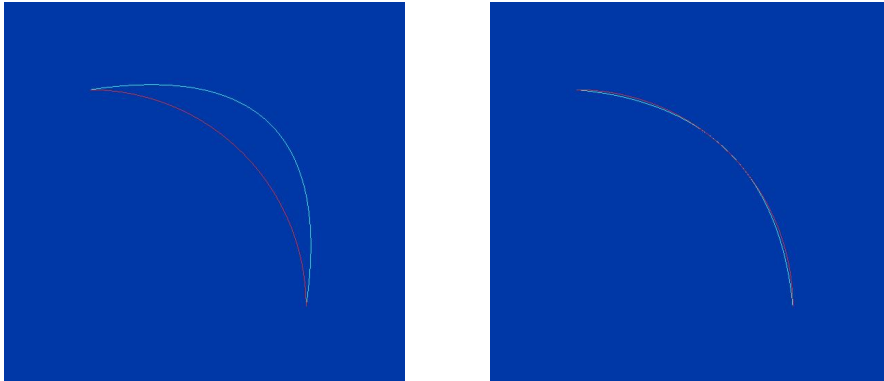


Figure 2: The CAD designed arc and the Geometry Generated arc (after 5 BFGS iteration and 10 BFGS iterations)

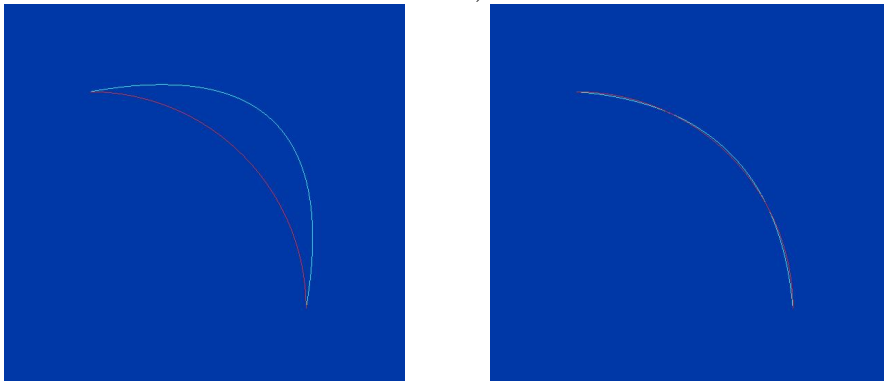


Figure 3: The CAD designed arc and the Geometry Generated arc (after 2 LMDIF iteration and 5 LMDIF iterations)

## 5 Second step: Unlofting a simple plane

We now move on to the more complicated problem of unlofting the geometry of a simplified airplane which has no tails nor engines. In the following text we refer to this as the Simple Plane problem. The airplane designed by an engineer has six surface pieces: a body, a body cap, a starboard wing with a cap, and a port wing with a cap. Our geometry generator creates similar planes with the same number of surface pieces using the eighteen preset parameter values, as seen in Figure 4. The parameter descriptions along with their upper and lower bounds and default values are shown in Table 5. Note that the only dimensional parameter is `body.length`. All the other parameter values are ratios or dimensionless variables which are then scaled according to `body.length` within the program. The parameter `zposition` is an absolute value.

The first task with the baseline configuration of the simple plane is to move it to the same coordinate system as the plane produced by the geometry generator. This consists of translating the plane and ensuring the rotation is correct. Next we match the surfaces of the baseline configuration plane with those of the geometry generator since we are not ensured of the same number of surfaces on both planes.

After running our routine several times and trying to eliminate the glaring programming errors, we find that the wings of the geometry generator plane are not matching in size to the baseline configuration plane. The error is due to the fact that we have not sampled a point on the wing cap and therefore there is no boundary being enforced as the points are projected. We correct this oversight and our preliminary attempts at producing the parameter values for the geometry generator according to the baseline configuration do yield a residual vector sum on the order of  $1.0E-4$ . On comparing the parameter set with the actual answer, we are wrong on three out of the eighteen. Table 5 and the graph in Figure 5 summarize the number of iterations and the sum of the residual vector that result from our miscalculations on the simple plane. While running our routine, *Geoduck* at times either returns an error or shuts down completely. In order to save our previous steps, we are required to output every vector calculated. We use this vector as a new initial guess and restart the nonlinear solver routine. The charts, therefore, differentiate between the number of times we have had to restart the routine, called a run, and the number of iterations used by the nonlinear solver.

Our incorrect results from the first try lead us to conclude that a small

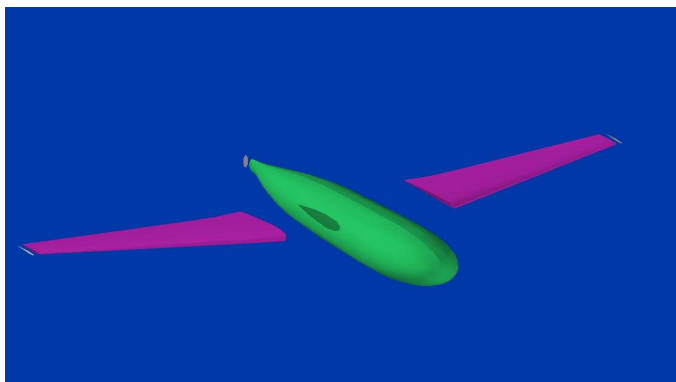


Figure 4: The surfaces of both the baseline configuration and the geometry generator are the same and consist of a body, body cap, starboard and port wings and their associated caps.

residual vector sum is not sufficient proof of the convergence of the routine to the correct parameter set solution. A closer examination of the physical appearances of the two planes exposes the error: we have not sampled a sufficient number of points within the Yehudi span of the wing. We therefore sample more points along the wing and did, in the end, produce the correct initial parameter set values. We record the change in distance over the iterations in Figure 6 and list the number of iterations in Table 5.

Table of the simple plane parameter values, the initial guess, and the final result.

Table summarizing the number of runs required and the number of iterations the nonlinear solver used to solve the simple plane problem. This table contains the results that were erroneous, that is, the Yehudi values were not correct.

Table summarizing the number of runs required and the number of iterations the nonlinear solver used to solve the simple plane problem.

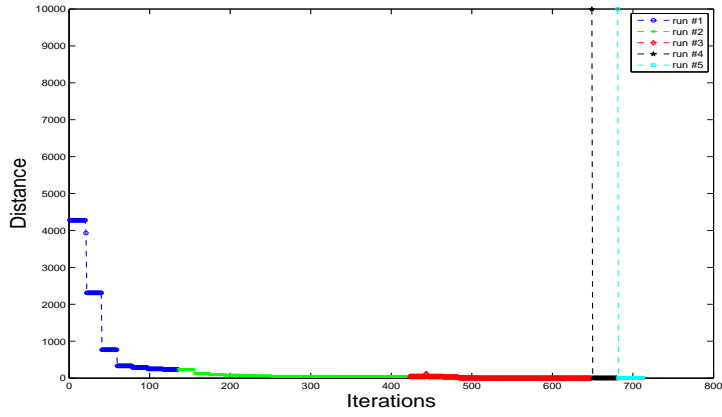


Figure 5: Graph of distances the nonlinear solver returned as it iterated on the Simple Plane problem. These are the results for the Simple Plane with the incorrect Yehudi values.

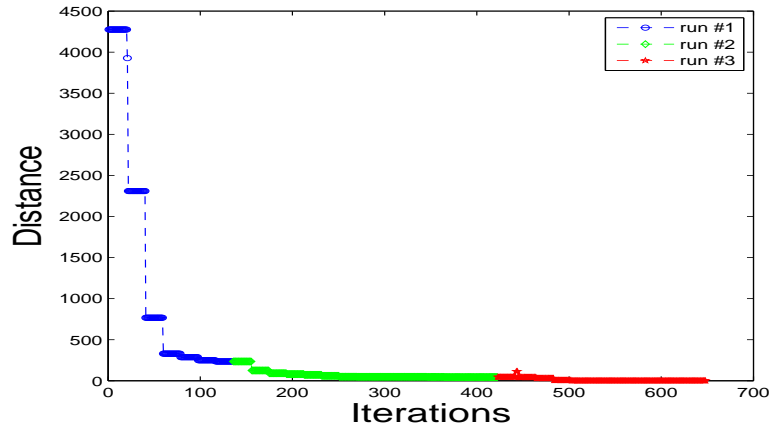


Figure 6: Graph of distances the nonlinear solver returned as it iterated on the Simple Plane problem.

Parameter Name	Lower Bound	Upper Bound	Default Value	Final Value
body.middlearea	0.01	$\infty$	0.05	0.043
body.endarea	0.01	$\infty$	0.025	0.031
body.length	100	$\infty$	500.0	622.0
body.blade_aspect	0.01	100	4.0	3.75
body.taper_start	0.25	0.9	0.5	0.375
wing.root_toverc	0.05	0.3	0.17	0.17
wing.taper_ratio	0.01	1.0	0.35	0.35
wing.aspect_ratio	3.0	15.0	13.5	8.0
wing.quarterchord_sweep	-10.0	40.0	7.0	17.0
wing.tip_toverc	0.05	0.3	0.16	0.1
semispan	0.2	$\infty$	0.8	0.7
wing.dihedral	-20.0	20.0	0.0	0.0
wing.le_kappa	$-\infty$	0.0	-4.0	-2.5
wing.quarterchord_kappa	$-\infty$	0.0	-4.0	-3.5
wing.yehudi_span	0.05	0.95	0.3	0.25
wing.yehudi_root	0.0	1.0	0.1	0.05
wingposition	0.1	0.7	0.47	0.25
zposition	-30.0	30.0	30.0	26.0

Table 1: Simple plane summary of parameters

	Solver Iterations	Distance
Initial		4276.518534983301
First Run	null	234.95206888264877
Second Run	288	44.60536464596181
Third Run	225	1.6924027350745887E-4
Fourth Run	30	1.6924027350745887E-4
Fifth Run	30	1.6924027350745887E-4

Table 2: Behavior of distance compared to iteration numbers and restarts

	Solver Iterations	Distance
Initial		4276.518534983301
First Run	null	234.95206888264877
Second Run	288	44.60536464596181
Third Run	225	1.6924027350745887E-4

Table 3: Behavior of distance compared to iteration numbers and restarts

## 6 Third step: Unlofting a harder plane

After solving the problem with the simple plane, we use the same base algorithm to solve the harder plane problem. Instead of receiving a plane with the same geometry as those produced by the included geometry generator, we had a different number of surfaces. As shown in Figure 7, the baseline configuration wing consists of five pieces total whereas in the geometry generator plane shown in Figure 8, the wing consists of only two pieces: the wing surface and the wing cap. Another visibly noticeable difference is that the baseline configuration produces only half of a plane. Because of this we have to pick the sample points accordingly. The geometry generator for the harder plane has 34 parameter values and they are listed in Table ??.

Instead of producing a statement to loop over the surfaces of the plane as in the simple plane problem, we have to take care when projecting the points from the different wing sections of the harder plane baseline configuration to ensure that they are distributed to the correct sections of the geometry generator plane. We sample points on all the sections of the wing, minus the wing cap, and combine these points all onto one list. We project this merged list onto the wing of the geometry generator plane within the objective function.

Due to the time constraint of a limit of 10 days, we were unable to produce the desired results for the harder plane problem. After several iterations and runs, the minimum residual vector sum that we were able to achieve is on the order of  $1.0E1$  but this is a significant achievement from the initial residual sum on the order of  $1.0E3$ . The iterations and the number of runs are in Figure 9 and in Table 6.

Table summarizing the number of runs required and the number of iterations the nonlinear solver used to solve the harder plane problem.

	Solver Iterations	Distance
Initial		9632.761039388799
First Run	null	470.8346231418708
Second Run	2052	97.29840180491314
Third Run	681	8.668994052115943

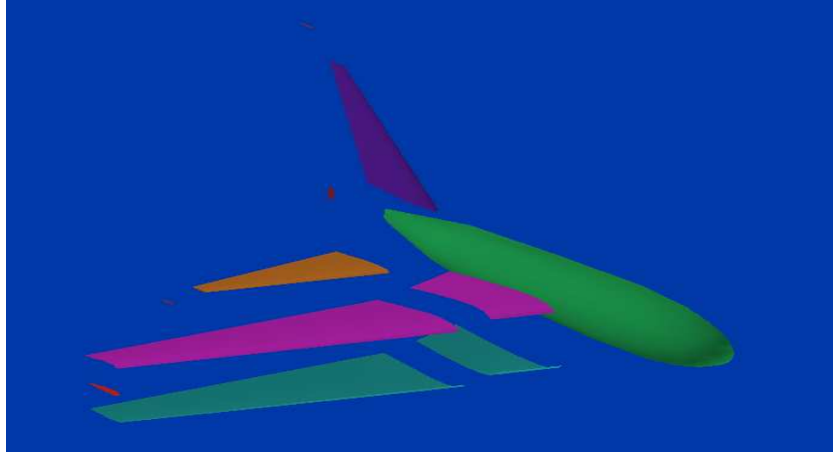


Figure 7: The surfaces of the baseline configuration for the harder plane problem.

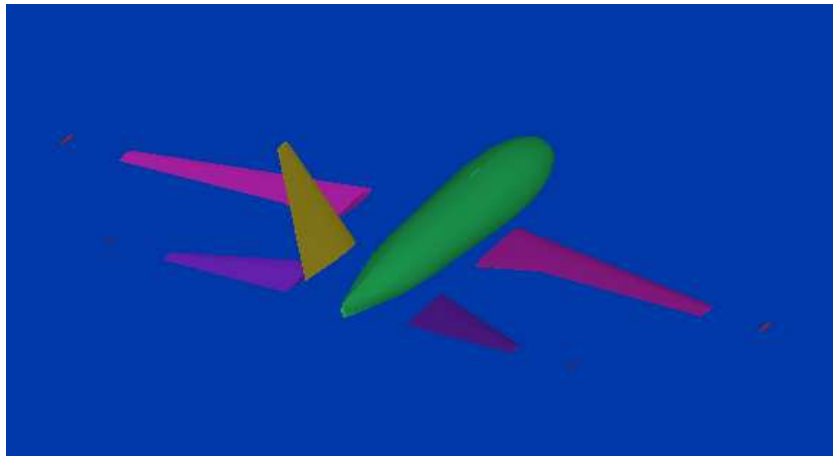


Figure 8: The surfaces of the geometry generator plane for the harder plane problem.

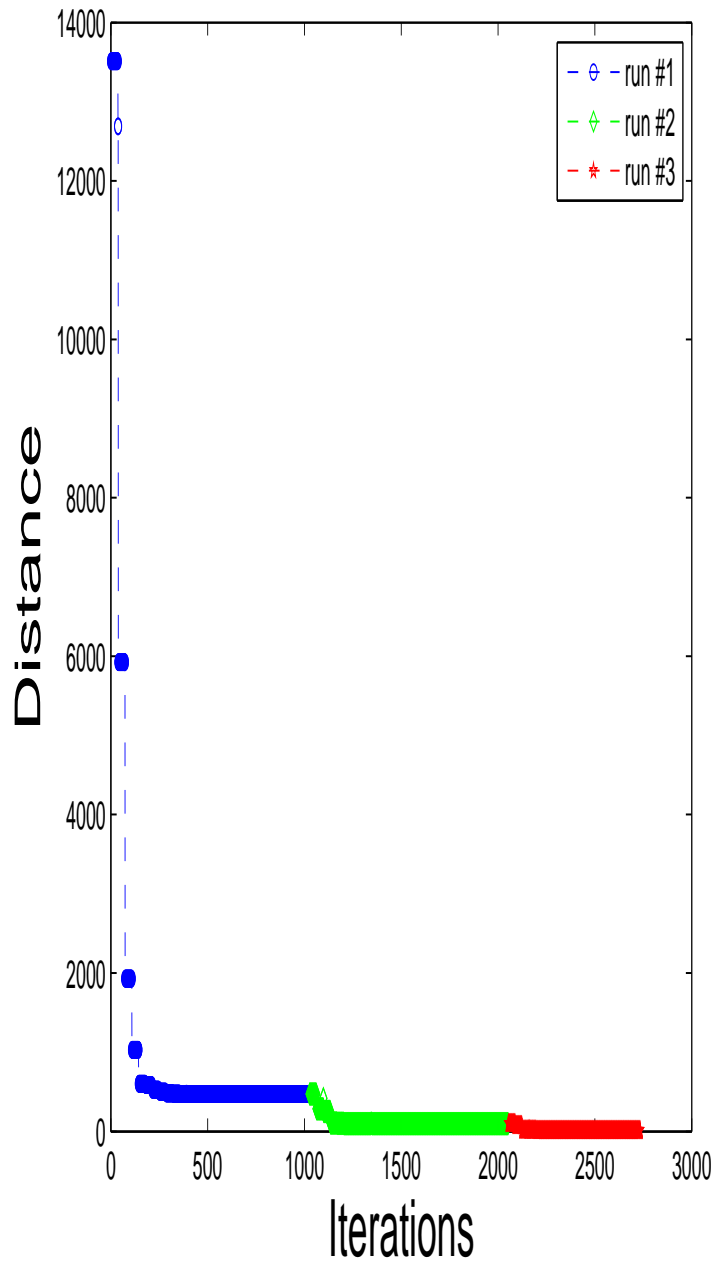


Figure 9: Graph of distances the nonlinear solver returned as it iterated on the harder plane problem.

Parameter Name	Lower Bound	Upper Bound	Default Value	Final Value
body.middlearea	0.01	100000	0.05	0.042999984
body.endarea	0.01	100000	0.025	0.021
body.length	100	1357	1257	1257
body.blade_aspect	0.01	100	4	3.149999981
body.taper_start	0.25	0.9	0.5	0.529999997
wing.root_toverc	0.05	0.3	0.17	0.173018224
wing.taper_ratio	0.01	1	0.35	0.348742576
wing.aspect_ratio	3	15	13.5	8.976782948
wing.quarterchord_sweep	-10	40	7	18.18518158
wing.tip_toverc	0.05	0.3	0.16	0.164162248
wing.semispan	0.2	100000	0.8	0.829995522
wing.dihedral	-20	20	0	6.997381856
wing.le_kappa	-100000	0	-4	58.26293563
wing.quarterchord_kappa	-100000	0	-4	4.117484285
wing.yehudi_span	0.05	0.95	0.3	0.224279872
wing.yehudi_root	0	1	0.1	0.079811308
wing_xpos	0.1	0.7	0.47	0.320007237
wing_zpos	-0.3	0.3	-0.04053179	-0.044958471
htail.root_toverc	0.05	0.3	0.17	0.173375824
htail.taper_ratio	0.01	1	0.35	0.25008208
htail.aspect_ratio	3	15	13.5	7.252995659
htail.quarterchord_sweep	-10	40	7	19.542545
h_semispan	0.2	100000	0.4	0.35499998
htail.dihedral	-20	20	0	4.999974209
htail.le_kappa	-100000	0	-4	-207.3418146
htail_xpos	0.1	0.9	0.8	0.775672376
htail_zpos	-0.9	0.9	0	2.12E-07
vtail.root_toverc	0.05	0.3	0.17	0.118960428
vtail.taper_ratio	0.01	1	0.35	0.197348247
vtail.aspect_ratio	3	15	13.5	5.501674456
vtail.quarterchord_sweep	-10	40	7	30.01622052
v_semispan	0.2	100000	0.51	0.340015584
vtail.le_kappa	-100	0	-4	-133.4689594
vtail_xpos	0.6	0.9	0.7	0.759925649

Table 4: Table of the harder plane parameter values, the initial guess, and the final result.

## 7 Conclusions

The work produced during our time at the IMA Mathematical workshop just scratched the tip of the surface of a much harder problem at Boeing. Future work involves designing a completely automatic unlofting procedure that includes automatic sampling strategies, automatic boundary calculations, and automatic surface ordering. The loft-free unlofting problem includes automatically producing a geometry generator. For progress to be made in the loft-free problems, much research still needs to be conducted.

## 8 Acknowledgements

We would like to thank Thomas A. Grandine and the Boeing company for providing this research problem for us and being our mentors during our stay here. Thank you for all of your patience and teaching that you gave us. We extend much gratitude as well to the IMA for organizing this workshop and inviting us to attend.

## References

- [1] R.Fletcher, Practical Methods of Optimization, John Wiley & Sons Ltd, Chichester, 1987
- [2] J.E. Dennis, Jr., Robert **B.** Schnabel, Numerical Methods for Unconstrained Optimization and Nonlinear Equations, Siam, Philadelphia, 1996