Project 1: Aircraft Design and Performance

Dylan Bryant

1. Introduction

For Task 1, I was tasked with finding the total lift, drag and lift to drag ratio created by a wing given the pressure and shear stress at sensors placed on its perimeter. Using given formulas, that needed to be implemented into MATLAB code. The script also needed to be able to print these values for the user to read.

Task 2 required me to create a MATLAB function capable of calculating total drag based on the aspect ratio (AR) and total wing area (S). After creating the function, the script needed to be able to differentiate the function with respect to aspect ratio and wing area based on 4 different approximation methods and an additional “exact” method using imaginary numbers and an extremely small Δx. The four approximation methods were compared to the exact derivative to calculate, then graph, the error in each method for both the derivative of drag with respect to each variable. The magnitude of Δx and value of the derivative were printed at the “best Δx” (least error). Next the function needed to be ran through an optimization function to find the best aspect ratio and wing area combination to minimize the total drag. The optimal aspect ratio and wing area were then printed, along with the respective drag and lift to drag ratio at the optimal values.

2. Methods

For Task 1, the sensor data needed to be converted into useful forces instead of pressures. The net force on the wing could be broken down as

$$\overrightarrow{F'} = \int_{airfoil} \left(p\vec{n} + \tau_w \vec{t}\right) ds \qquad (1)$$

where p is the pressure, n is the normal vector, τ is the shear stress and t is the tangential vector.

Because this equation would be difficult to materialize in MATLAB, an alternate but similar approach was used. Integrals are similar to summations, so a sum was used in its place, and instead of calculating the net force, the force in the X and Y directions were calculated separately. This was done by using a tilted coordinate system. Pressure was used as Y’ and shear stress was used as X’ where the force in the X direction is drag and the force in the Y direction is lift. The angle between the coordinate systems was calculated by taking the arctangent of the difference in Y values over the difference of X values of midpoints between the observed sensor and its adjacent sensors. The angle is then adjusted to account for the angle of attack of the wing.

Using geometric equations found from the tilted coordinate plane, the pressure and shear stress were broken down into their X and Y components, multiplied by the distance

they acted over (distance between the midpoints) and summed to generate lift and drag values. This was done in a loop that repeated for every sensor. The loop contained conditional statements for the first and last sensor, which only had 1 adjacent sensor to use in the midpoint calculation. The position of the tip of the airfoil, located on the trailing edge, was used in leu of another sensor in both cases.

Using given values for constants, the Coefficient of Lift and Drag were both calculated and then used to calculate the Lift to Drag ratio. The ratio was verified by calculating it using magnitudes of lift and drag was well.

Task 2 was significantly more complex than task 1. The drag function, which is used throughout the task, was created in Question 1. This function allows for the input of the aspect ratio and surface area of the wing, as well as a struct that carries values for the constants related to calculations. The function begins by calculating all variables that are based on the aspect ratio or wing area, such as the length and width of the wing and the wetted surface area. Using those algebraically calculated values and constants, the function then calculates the total weight of the aircraft using an iterative method. After the weight has been calculated, the coefficients are able to be calculated. Using the coefficients and other pre-calculated constants, the drag and lift to drag ratio are calculated and outputted by the function.

Question 2 of Task 2 was preformed mostly on paper and was a derivation of the Fourth Order Central numerical differentiation scheme. The Taylor Table used to assist in the derivation is shown in Figure 1, as well as the algebra used to set up matrix A and b. MATLAB code was used to solve the matrix equation. The values in the x matrix are the values of the coefficients of the different values of the function at different times. This method of numerical differentiation, as well as 3 other methods, are used in question 3 to calculate the derivative of the drag function.

At first, Question 3 of Task 2 took the most lines of code to complete. An array of values at constant multiples away from each other between the values of 10^{-20} and 1 was created first. Values in the array were plugged into variations of the drag function to get derivatives at different accuracies. In order to calculate the derivatives, values of the function were needed from $f(x-3\Delta x)$ to $f(x+3\Delta x)$. These were calculated for the aspect ratio = x and the wing area = x. Those values are plugged into the given equations for numerical differentiation to achieve an approximate derivative. Using the complex step method, which source code was given for, the exact value of the derivative with respect to both variables was calculated. The absolute value of the difference between the derivative at each Δx size was calculated and graphed for both variables and all 4 numerical differentiation methods. Figure 2 shows the error plots for the approximations based on the exact derivative.

Later, I changed my code to provide identical results in a much more compact way for most of the differentiation methods. Using the source code provided for complex step, I added more cases that computed the First Order Forward, Second Order Central, and the Fourth Order Central. It evaluates the derivatives the same exact way mathematically, but the lines of code needed to calculate the values in my main scrip dropped from over 50 to

just 8 (excluding Pade which remained evaluated in the large loop). I left the less condensed method in my main function to demonstrate the fact that there are multiple ways of achieving the same results.

An optimization code was used for Question 4. In order to use the optimization code, the drag function had to be edited to accept a matrix that contained values for the aspect ratio and the wing area because the optimization function only allowed for one input, but that input could be a matrix. Lower bounds are set at a very small, positive, non-zero value to prevent NaN or Inf errors in code that would cause it to run indefinitely. An additional function that contains the constraints of the function was also created for use by the function. It is used by the optimization function to ensure the values it is testing are sensible in a real-world scenario. The function also contained an edit that allowed for derivatives to be input to create a gradient. The gradient is used to speed up the optimization process by giving the optimization function data for the slope to nudge the tested values in the correct direction. When the derivatives are used in the optimization code, it yields a slightly different result, but I am unsure why that is.

3. Questions and Tasks

3.1 Task 1: Sectional Lift-to-Drag Ratio

i Q1.

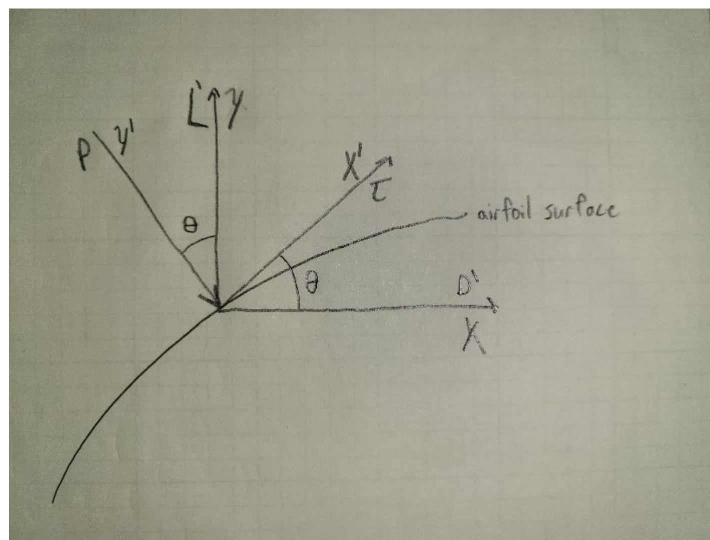


Figure 1. Tilted Coordinate System on sample point of airfoil.

This question was completed using a loop that summed the sectional lift and drags at each sensor. To do so, the tilted coordinate system method (Shown in Figure 1) was used instead of vectorizing, as any vectorization would theoretically result in the same values for the final answer. The midpoints were calculated by taking the positions of the adjacent sensors and averaging them with the position of the observed sensor.

$$xmid_{forward} = \frac{xpos_i + xpos_{i+1}}{2} \tag{2}$$

$$xmid_{behind} = \frac{xpos_i + xpos_{i-1}}{2} \tag{3}$$

$$ymid_{forward} = \frac{ypos_i + ypos_{i+1}}{2} \tag{4}$$

$$ymid_{behind} = \frac{ypos_i + ypos_{i-1}}{2} \tag{5}$$

Using geometric relationships, the theta from horizontal was then calculated to adjust the pressure and shear stresses later. Theta from horizontal was found with

$$\theta = \arctan\left(\frac{\Delta y}{\Delta x}\right) = \arctan\left(\frac{ymid_{forward} - ymid_{behind}}{xmid_{forward} - xmid_{behind}}\right) \tag{6}$$

The distance between the midpoint was calculated by Pythagorean

$$ds = \sqrt{\left(xmid_{forward} - xmid_{behind}\right)^2 + \left(ymid_{forward} - ymid_{behind}\right)^2} \tag{7}$$

The values found in Equations 6 and 7 are used in Equation 8 and 9 to calculate Lift and Drag respectively

$$Lift = (-Pcos(\theta - \alpha) + \tau \sin(\theta - \alpha))ds \tag{8}$$

$$Drag = -(Psin(\theta - \alpha) + \tau \cos(\theta - \alpha))ds \tag{9}$$

where α is the angle of attack of the airfoil. The lift and drag values for each point are stored and summed after the loop has completed.

All of this is done in order to calculate the Coefficient of Lift (C_l) and the Coefficient of Drag (C_d). The C_l and C_d are calculated with

$$C_l = \frac{L}{qc} \tag{10}$$

$$C_d = \frac{D}{qc} \tag{11}$$

where $q = \frac{1}{2}\rho U_\infty^2$ where ρ is air density and U_∞ is the freestream velocity.

Numerical values of C_l and C_d were found to be 0.805632 and 0.068891 respectively by use of the MATLAB code. Those values create a Lift-to-Drag ratio of 11.694271

3.2 Task 2: Minimizing Drag

i Q1.

A function was created that accepted 3 inputs – the aspect ratio, the wing area, and a struct containing values for all the used constants. Its outputs are the total drag on the aircraft, C_l, and C_d. After unpacking the struct into variables that were easier to call later in the function, other constants that needed to be calculated were evaluated based on the given constant values. Out of the 5 unknown values needed to calculate Drag, C_l, and C_d, 4 were able to be calculated algebraically – Wingspan (b), Chord Length (c), Reynolds Number, Skin Friction Coefficient (C_f).

$$b = \sqrt{AR * S} \tag{12}$$

$$c = \frac{S}{b} \tag{13}$$

$$Re = \frac{\rho U_\infty c}{\mu} \tag{14}$$

where μ is the dynamic viscosity of air,

$$C_f = \frac{0.07}{Re^{\frac{1}{6}}} \tag{15}$$

The one value it was not possible to calculate algebraically was the total weight of the aircraft. The problem statement provided a value for the weight of the aircraft without its wing, but the weight of the wings needed to be calculated each time the function was run as it is dependent on the values of AR and S (as well as some of the intermediate values such as b and c). To solve for the total weight of the aircraft, an iterative method was used. Using Equations 16 and 17, Equation 18 was formed.

$$W = W_w + W_o \tag{16}$$

$$W_w = a_1 S + \frac{a_2 N_{ult} b^3 \sqrt{W_o W}}{S(\frac{t}{c})} \tag{17}$$

$$W_w = a_1 S + \frac{a_2 N_{ult} b^3 \sqrt{W_o (W_w + W_o)}}{S(\frac{t}{c})} \tag{18}$$

where W_w is the weight of the wings, W_o is the weight without the wings, W is the total weight of the aircraft, a_1 is the first empirical coefficient in the wing sizing, a_2 is the second empirical coefficient in the wing sizing, N_{ult} is the ultimate load factor, and $\frac{t}{c}$ is the average thickness-to-chord ratio. An initial guess was provided for W_w, then evaluated. If the value of the output of Equation 18 was within 10^{-16} (double machine precision), the loop was broken and an output provided. If it was not within the tolerance, the output of

Equation 18 was used as a new input and the loop was run again. Using Equation 16, the total weight was calculated after the error was within the bounds.

Having the total weight allowed for the C_l to be calculated by

$$C_l = \frac{W}{qS} \tag{19}$$

which provided enough information to calculate

$$C_d = \frac{kC_f S_{wet}}{S} + \frac{C_l^2}{\pi e AR} + \frac{a_3}{S} \tag{20}$$

where k is the form factor, S_{wet} is the wetted wing area and a_3 is the empirical coefficient in C_d. Using this calculated C_d, the total drag can be calculated

$$D = qcC_d \tag{21}$$

With drag being calculated, and C_l and C_d being calculated as intermediate values, all outputs of the function are able to be calculated.

Drag was evaluated to be 516.7642 Newtons, with the Coefficient of Lift being 1.0360 and the Coefficient of Drag being 0.0551 at AR = 10 and S = 25.

The equations used to calculate Drag are the best equations to use for a double precision machine since exact numbers are used. Because all of the numbers in the equation are exact, this means that the answer would be accurate to the full 16 decimal places a double precision machine is computing. If you were to simplify the equations using constants prior to their use in the drag calculating function, there is a possibility of higher rounding error that can negatively impact the results.

ii Q2.

This question asked to derive the Fourth order Central numerical differentiation scheme using a Taylor Table. The Taylor Table I constructed is shown below in Table 1.

	f_j	f_j'	f_j''	f_j'''	$f_j^{(4)}$	$f_j^{(5)}$
f_j'	0	1	0	0	0	0
$a_{-2} f_{j-2}$	a_{-2}	$-2a_{-2}(\Delta x)$	$a_{-2}\frac{(2\Delta x)^2}{2}$	$-a_{-2}\frac{(2\Delta x)^3}{6}$	$a_{-2}\frac{(2\Delta x)^4}{24}$	$-a_{-2}\frac{(2\Delta x)^5}{120}$
$a_{-1} f_{j-1}$	a_{-1}	$-a_{-1}(\Delta x)$	$a_{-1}\frac{(\Delta x)^2}{2}$	$-a_{-1}\frac{(2\Delta x)^3}{6}$	$a_{-1}\frac{(2\Delta x)^4}{24}$	$-a_{-1}\frac{(2\Delta x)^5}{120}$
$a_0 f_j$	a_0	0	0	0	0	0
$a_1 f_{j+1}$	a_1	$a_1(\Delta x)$	$a_1\frac{(\Delta x)^2}{2}$	$a_1\frac{(2\Delta x)^3}{6}$	$a_1\frac{(2\Delta x)^4}{24}$	$a_1\frac{(2\Delta x)^5}{120}$
$a_2 f_{j+2}$	a_2	$2a_2(\Delta x)$	$a_2\frac{(2\Delta x)^2}{2}$	$a_2\frac{(2\Delta x)^3}{6}$	$a_2\frac{(2\Delta x)^4}{24}$	$a_2\frac{(2\Delta x)^5}{120}$

This table can be broken down into

$$(a_{-2} + a_{-1} + a_0 + a_1 + a_2)f_j + (1 + 2a_{-2}\Delta x - a_{-1}\Delta x + a_1\Delta x + 2a_2\Delta x)f_j' + (4a_{-2}\Delta x^2 + a_{-1}\Delta x^2 + a_1\Delta x^2 + 4a_2\Delta x^2)f_j'' + (-8a_{-2}\Delta x^3 - a_{-1}\Delta x^3 + a_1\Delta x^3 + 8a_2\Delta x^3)f_j''' + (16a_{-2}\Delta x^4 + a_{-1}\Delta x^4 + a_1\Delta x^4 + 16a_2\Delta x^4)f_j'''' = O(\Delta x^4) \quad (22)$$

which becomes the matrix equation

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & 0 & 1 & 2 \\ 4 & 1 & 0 & 1 & 4 \\ -8 & -1 & 0 & 1 & 8 \\ 16 & 1 & 0 & 1 & 16 \end{bmatrix} \begin{bmatrix} a_{-2} \\ a_{-1} \\ a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{\Delta x} \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (23)$$

that can be solved in MATLAB to yield $a_{-2} = \frac{1}{12}$, $a_{-1} = \frac{-8}{12}$, $a_0 = 0$, $a_1 = \frac{8}{12}$, $a_2 = \frac{-1}{12}$ which when plugged back into the left most column of the Taylor Table, yields

$$f_j' = \frac{(f_{j-2} - 8f_{j-1} + 8f_{j+1} - f_{j+2})}{12\Delta x} + O(\Delta x^4) \quad (24)$$

Which is our equation for the Fourth order Central method of numerical differentiation and provides fourth order accuracy based on the four evaluation points of the function per derivative.

iii Q3.

Question 3 uses four numerical differentiation approximations, as well as a 'complex step' exact derivative to analyze the derivative of the drag function created in Q1 since it is impossible to find a derivative with respect to the inputs with standard methods. The four differentiation methods used are

$$f_j' = \frac{f_{j+1} - f_j}{\Delta x} + O(\Delta x) \quad (25)$$

$$f_j' = \frac{(f_{j+1} - f_{j-1})}{2\Delta x} + O(\Delta x^2) \quad (26)$$

$$\begin{aligned} f_1' + 2f_2' &= -\frac{5}{2\Delta x}f_1 + \frac{2}{\Delta x}f_2 + \frac{1}{2\Delta x}f_3 + \mathcal{O}(\Delta x^3) \\ f_{j-1}' + 4f_j' + f_{j+1}' &= \frac{3f_{j+1} - 3f_{j-1}}{\Delta x} + \mathcal{O}(\Delta x^4),\ j = 2, 3, \cdots, N-1 \\ f_N' + 2f_{N-1}' &= \frac{5}{2\Delta x}f_N - \frac{2}{\Delta x}f_{N-1} - \frac{1}{2\Delta x}f_{N-2} + \mathcal{O}(\Delta x^3), \end{aligned} \quad (27)$$

and Equation 24, where Equation 25 is the First order Forward difference method, Equation 26 is the Second order Central difference method, and Equation 27 is Pade. All four numerical methods were evaluated with 1000 or more logarithmically spaced values for Δx between 10^{-20} and 10^{0}. The derivatives were calculated with respect to

AR and S using the numerical and exact methods, except Pade, using a case switching code that is a modified version of the complex step code provided.

Two copies of the code were made, one for the derivative of Drag with respect to AR and one with respect to S. Both functioned the same way and had the same inputs and outputs. The function required the input for your Point of Interest, which is the value you would like to find the derivative at, the struct containing constants, to pass through to the drag function, and the method of differentiation, which told the code what method to use in the case switch. The Points of Interest (PoI) were at the same values from Q1, AR = 10 and S = 25. For all methods except complex, the function would calculate the derivative at all of the Δx values and store those values for use later. The code was laid out identically to the numerical differentiation methods listed in Equations 24-27. The values for f in the equations were substituted with the values of drag calculated by the drag function at the Δx being observed. (For f_{j+1}, it was evaluated at the PoI+Δx, for f_{j+2}, the function was evaluated at PoI+($2\Delta x$), and so on). After running the code with respect to AR and S, eight matrices were created that contained the values of the derivatives at each step size.

I could not find a reliable way to incorporate the Pade method into my modified version of the source code, and have previously found a way to not use the source code when I was struggling to determine how the source code functioned, so I elected to use my from-scratch code for that portion. The A matrix for the Pade method was set up first. It was set up in a loop of changeable size to be adapted to higher accuracies if needed. The b matrix was formed in a loop that ran for the same length as the matrix containing the Δx values and consisted of Equation 27 repeated to N = 7 The accuracy of the b matrix is unable to be changed, so Pade is only able to be evaluated for a mesh size of 7 as my code stands. The x matrix is then solved for using linear algebra techniques embedded in MATLAB. To get a more accurate result, the Pade method was modified slightly to make f_1 actually equal to f_{j-3}. This makes f_j equal to f_4, so our derivative will be the central value in the x matrix after solving. All of the methods are evaluated against the exact derivative found using sample code provided, and the absolute error is plotted over the Δx to create a visual representation of accuracy. As seen in Figures 2 and 3 or any Δx lower than 10^{-16}, the error is the absolute value of the exact derivative due to the limitation of double precision computing. There is an “explosion region” for values that are still extremely small compared to the PoI where data is not consistent but is trending to be more accurate. As values for Δx get larger, the methods begin to have a linear slope until the end of the plot.

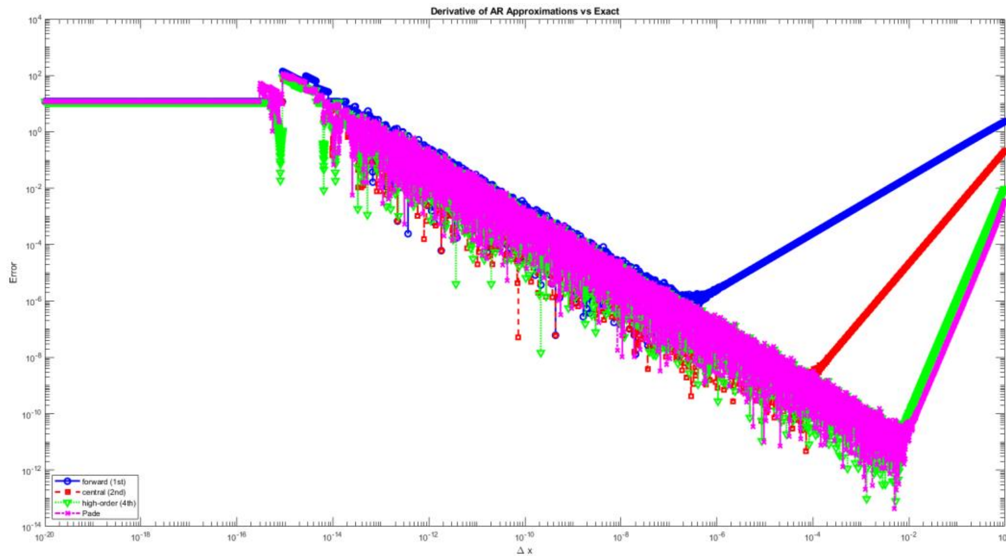


Figure 2. Error of dD/dAR over step size.

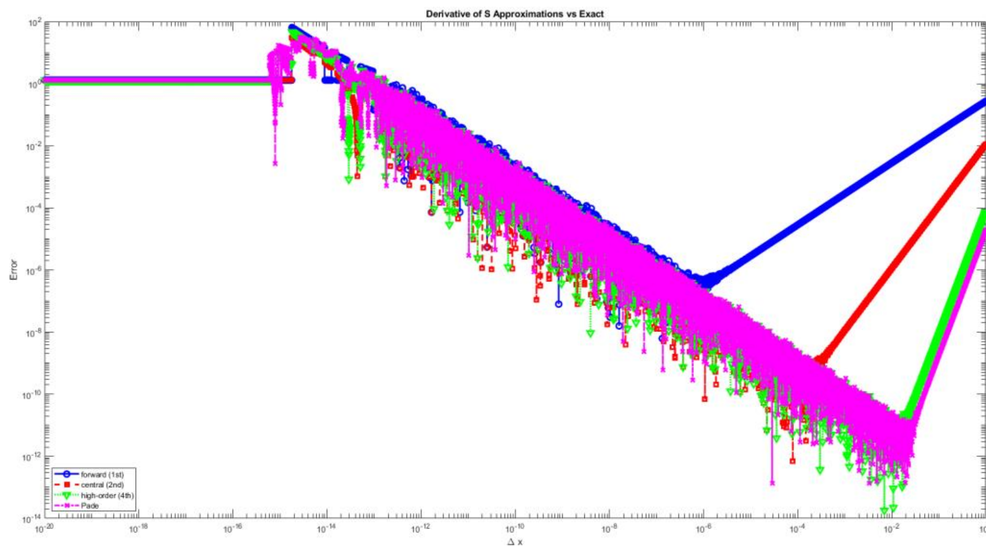


Figure 3. Error of dD/dS over step size.

The Fourth order Central method reaches the higher overall accuracy, followed closely by Pade, then Second order Central, and finally First order Forward with the lowest overall accuracy. This is to be expected given the orders of accuracy shown in the base equations. This can be verified by finding the slopes of the linear portions of the graphs. Fourth order Central and Pade share a very similar slope of about 4, Second order Central has a slope of 2 and First order Forward has a slope of 1. With the exception of Pade, the slopes of the convergence lines are the same as the methods order of accuracies. I am unsure why the slope for the Pade method does not match the order of accuracy, but it may

be related to the size of the mesh used in calculating. The best Δx, the derivative at that Δx, and the number of significant figures the derivate based on the exact is accurate to, with respect to each variable is listed in Tables 1 and 2 below

	With Respect to AR (exact: -11.768696415646703)			
Method	First order Forward	Second order Central	Fourth order Central	Pade
Best Δx	$1.5*10^{-7}$	$7.2*10^{-5}$	$5.3*10^{-3}$	$5*10^{-3}$
Derivative	-11.768696 404708983	-11.768696 415641957	-11.768696 415646623	-11.768696 415646746
Accurate Significant Figures	9	13	14	15

Table 1. Table of requested values for dD/dAR

	With Respect to S (exact: -1.299169779074778)			
Method	First order Forward	Second order Central	Fourth order Central	Pade
Best Δx	$1.3*10^{-7}$	$7.9*10^{-5}$	$6.9*10^{-3}$	$3.4*10^{-3}$
Derivative	-1.2991697 72874502	-1.2991697 79074089	-1.2991697 79074760	-1.2991697 79074645
Accurate Significant Figures	9	13	14	13

Table 2. Table of requested values for dD/dS

iv Q4.

Question 4 moves away from assessing the derivatives and more towards putting them to use. The Drag function was optimized using a MATLAB toolbox for optimization to find the best AR and S to result in the least total drag. Lower bounds must be defined at a positive, non-zero value to prevent unrealistic negative values as well as prevent NaN or Inf errors that cause the function to run indefinitely. Initial guesses must be provided, and the inputs for those were arbitrary given a limited range that would be enforced by the constraints. There were two constraints on the optimization function to relate it to a real-world scenario:

$$b \leq 20 \tag{28}$$

$$S \geq \frac{W}{\frac{1}{2}\rho U_{land}^2 C_{l,\max}} \tag{29}$$

using Equation 12, Equation 28 becomes

$$\sqrt{AR * S} - 20 \leq 0 \quad (30)$$

which can be used along with Equation 29 in another iteration of the drag function that evaluates if those inequalities are true. If they are, and the optimization function concludes the values for AR and S are at their lowest, the optimization function will provide a matrix of [AR_{opt},S_{opt}] as an output. The optimal values with no gradients input into the optimization function are AR = 13.4396 and S = 23.95, which produces a drag of 499.3378 Newtons. The optimization with no gradient resulted in a C_l of 1.2005 which is lower than the 3rd constraint contained in the problem statement of $C_{l,max}$ = 2 I actually found it detrimental to the optimization codes to input a gradient, as no matter what method I input, I would get an AR of 14.8996 and an S of 24.2282 which resulted in a drag of 502.2200 Newtons. I attempted to change the Δx in which the gradients were evaluated but no Δx produced a better result. Because of this, I will not be including the requested table containing the best Δx values as they produce irrelevant results. I am unsure as to why the optimal AR and S were different between the gradient and no gradient input optimization codes as they theoretically should have resulted in the same values, but the gradients would have assisted with the speed in which they were found.

4. Conclusion

There are multiple ways of solving for the total forces acting on a wing, as seen by the two very different methods used in the two tasks. While the first would rarely, if ever be used in a true real-world scenario it is possible to take raw data and turn it into meaningful results. Task two is a lot more likely to be encountered in the workplace but does take significantly more leg work to fully process the data, and that data can be solved for in a number of different way that all produce similar, if not the same results.

Similarly to the forces acting on the airfoils, the derivatives of functions can be found in multiple different, but still correct ways (pending how mission critical the results are). Out of the 5 ways to find the derivative of a function, it is apparent that the exact 'complex step' method is the most accurate as it is the baseline in which the others were judged. Fourth order Central and Pade both yield similar results at their most accurate points, and get to those points at similar speeds, but Pade is significantly more code therefore processing heavy. Fourth order Central provides extremely accurate results (to 14 significant figures in this case) with only a few lines of code being used for the derivative calculation. If the accuracy of results is not something of extreme importance, other methods such as Second order Central or First order Forward can be used for an even easier way of getting an approximation. Second order Central approximation was only 1 significant figure less accurate than the Fourth order Central for half the inputs. First order Forward requires 2 inputs, the same as Second order Central, and produced less accurate results – only accurate to 9 significant figures – so I do not see a reason to commonly use

this method. Second order Central would be a nice middle ground for most standard uses that include safety factors and other buffers for error in its values, with Pade or Fourth order Central being a solid option for cases where high accuracy results are needed and a function lacks an exact derivative.

Optimization code can also be tricky sometimes. The optimization function that was given the same constraints as all the rest, but less information about the function it was optimizing out preformed the code that was given high accuracy derivatives/gradients in this case. This highlights the need to verify the results of not just code, but also any other mathematical process, to make sure the outputs and reasonable and at least somewhat expected.

I am unaware of reasonable bounds for any of the results to be able to determine how reasonable my results are, but after conversation with my peers, my resultant values seem to be relatively average.

Table of Contents

Project 1 1
Task 2.1 1
Task 2.2 2
Functions 10

Project 1

dbryan19

```
clc
clear
close all
```

Task 2.1

Sectional Lift to Drag Ratio 2.1.1

```
%Constants:
p = 1.2; %Air Density in kg/m3
U1 = 40; %Freestream Velocity in m/s
U2 = 25;
u = 1.8*10^-5; %air viscocity in Ns/m2
a = 4; %angle of attack in degrees
c = 1; %chord legnth in m (total)
data = importfile("data.txt", [2, Inf]);
P = data.pressurePa; %Pressure in Pa
T = data.shearStressPa; %Shear in Pa
xpos = data.xm; %x values of sensors
ypos = data.ym; %y values of sensors
sensor = data.Sensor; %Used to find number of total sensors.
% Total number of sensors could be just written, but to make code
% "universial" this is useful
q = 0.5*p*(U1^2);
constants =
 struct('p',p,'U1',U1,'U2',U2,'u',u,'a',a,'k',1.2,'e',0.9,'Nult',2.5,'Wo',...

 6460,'a1',55.765,'a2',7.123*10^(-5),'a3',0.03589654,'t2c',0.12,'Uland',20,...
    'Clmax',2); %Assigns all variables to their numerical values in a
 struct

% Calculates the angle from horizonal of the tangent line and the
 distance
% between the midpoints (midpoint from observed sensor to forward
 sensor
% and observed to backwards sensor. Exceptions made for 1st and final
% sensor.
for i = 1:size(sensor,1)
    % Calculates the theta of the P-T plane vs the X-Y plane for all
```

```
    % sensors using adjacent midpoints, and also distance between
 midpoints
    if i>=2 && i<=size(sensor,1)-1
        xmid1 = (xpos(i-1)+xpos(i))/2;
        xmid2 = (xpos(i+1)+xpos(i))/2;
        ymid1 = (ypos(i-1)+ypos(i))/2;
        ymid2 = (ypos(i+1)+ypos(i))/2;
        theta = atand((ymid2-ymid1)/(xmid2-xmid1));
        dist = sqrt((xmid2-xmid1)^2+(ymid2-ymid1)^2);
    elseif i == 1
        xmid1 = (c+xpos(i))/2;
        xmid2 = (xpos(i+1)+xpos(i))/2;
        ymid1 = (ypos(i))/2;
        ymid2 = (ypos(i+1)+ypos(i))/2;
        theta = atand((ymid2-ymid1)/(xmid2-xmid1));
        dist = sqrt((xmid2-xmid1)^2+(ymid2-ymid1)^2);
    elseif i == size(sensor,1)
        xmid1 = (xpos(i-1)+xpos(i))/2;
        xmid2 = (c+xpos(i))/2;
        ymid1 = (ypos(i-1)+ypos(i))/2;
        ymid2 = (ypos(i))/2;
        theta = atand((ymid2-ymid1)/(xmid2-xmid1));
        dist = sqrt((xmid2-xmid1)^2+(ymid2-ymid1)^2);
    end
% Calculates lift and drag portion at each sensor
L(i) = (-P(i)*cosd(theta-a)+T(i)*sind(theta-a))*dist; %Stores lift
 values
D(i) = -(P(i)*sind(theta-a)+T(i)*cosd(theta-a))*dist; %Stores Drag
 values
end

Ltot = sum(L); %Calculates total Lift
Cl = Ltot/(q*c); %Calculates Lift Coeff
Dtot = sum(D); %Calculates totalDrag
Cd = Dtot/(q*c); %Calculates Drag Coeff
LtoD = Ltot/Dtot; %calculates Lift to Drag ratio based on total lift
 over total drag
LtoD2 = Cl/Cd; %for verification of results
% Prints out values
fprintf('The Coefficient of lift (Cl) is %f \nThe Coefficient of Drag
 (Cd) is %f \nThe Lift to Drag Ratio is %f\n\n\n'...
    ,Cl,Cd,LtoD)
```

The Coefficient of lift (Cl) is 0.805632
The Coefficient of Drag (Cd) is 0.068891
The Lift to Drag Ratio is 11.694271

Task 2.2

Minimizing Drag

```
% 2.2.1.1
```

```
%Constants: Most included in constants struct
AR = 10; %Aspect Ratio
S = 25; %Wing Area in m^2

[D,Cl,Cd] = drag(AR,S,constants);

% 2.2.1.2
A = [ 1 1 1 1 1
    -2 -1 0 1 2
    4 1 0 1 4
    -8 -1 0 1 8
    16 1 0 1 16];
b = [0 1 0 0 0]';
xTT = A\b;

% 2.2.1.3

format long
% Pade Setup
N = 7; %Number of points on mesh
A = zeros(N);
for i = 1:N %Makes A matrix for any N
    for j = 1:N
        if i - j == 1 && i ~= N
            A(i,j) = 1;
        elseif i - j == 1
            A(i,j) = 2;
        end
        if j - i == 1 && i ~= 1
            A(i,j) = 1;
        elseif j - i == 1
            A(i,j) = 2;
        end
        if i - j == 0
            A(i,j) = 4;
        end
        if (i == 1 && j == 1) || (i == N && j == N)
            A(i,j) = 1;
        end
    end
end

steps = 10000; %Number of iterations to calculate at
iter = ((logspace(0,-20,steps))); %Sets up iterations
bAR = zeros(size(iter,2),7);
bS = zeros(size(iter,2),7);


%The following code is only used for pade. It was written prior to
%understanding the source code given. Kept in the script to show there
 are
%multiple methods to solve for the derivatives.
for i = 1:size(iter,2)
    %Setup for use in approximation functions
```

```
    [Dn3,~,~] = drag(AR-(3*iter(i)),S,constants);
    [Dn2,~,~] = drag(AR-(2*iter(i)),S,constants);
    [Dn1,~,~] = drag(AR-iter(i),S,constants);
    [Dp1,~,~] = drag(AR+iter(i),S,constants);
    [Dp2,~,~] = drag(AR+(2*iter(i)),S,constants);
    [Dp3,~,~] = drag(AR+(3*iter(i)),S,constants);

    %First order Forward for dD/dAR
%     dDdARFoF(i) = double((Dp1 - D)/iter(i));

    %Second order Central for dD/dAR
%     dDdARSoC(i) = double((Dp1 - Dn1)/(2*iter(i)));

    %Fourth order Central for dD/dAR
%     dDdARFoC(i) = double((Dn2-(8*Dn1)+(8*Dp1)-Dp2)/(12*iter(i)));

    %Pade
    bAR(i,1) = double((-5*Dn3)/(2*iter(i)) + (2*Dn2)/(iter(i)) +
 (Dn1)/(2*iter(i)));
    bAR(i,2) = double(((3*Dn1)-(3*Dn3))/(iter(i)));
    bAR(i,3) = double(((3*D)-(3*Dn2))/(iter(i)));
    bAR(i,4) = double(((3*Dp1)-(3*Dn1))/(iter(i)));
    bAR(i,5) = double(((3*Dp2)-(3*D))/(iter(i)));
    bAR(i,6) = double(((3*Dp3)-(3*Dp1))/(iter(i)));
    bAR(i,7) = double((5*Dp3)/(2*iter(i)) - (2*Dp2)/(iter(i)) - (Dp1)/
(2*iter(i)));

    %Setup for use in approximation functions
    [Dn3,~,~] = drag(AR,S-(3*iter(i)),constants);
    [Dn2,~,~] = drag(AR,S-(2*iter(i)),constants);
    [Dn1,~,~] = drag(AR,S-iter(i),constants);
    [Dp1,~,~] = drag(AR,S+iter(i),constants);
    [Dp2,~,~] = drag(AR,S+(2*iter(i)),constants);
    [Dp3,~,~] = drag(AR,S+(3*iter(i)),constants);

    %First order Forward for dD/dS
%     dDdSFoF(i) = double((Dp1 - D)/iter(i));

    %Second order Central for dD/dS
%     dDdSSoC(i) = double((Dp1 - Dn1)/(2*iter(i)));

    %Fourth order Central for dD/dS
%     dDdSFoC(i) = double((Dn2-(8*Dn1)+(8*Dp1)-Dp2)/(12*iter(i)));

    %Pade
    bS(i,1) = double((-5*Dn3)/(2*iter(i)) + (2*Dn2)/(iter(i)) + (Dn1)/
(2*iter(i)));
    bS(i,2) = double(((3*Dn1)-(3*Dn3))/(iter(i)));
    bS(i,3) = double(((3*D)-(3*Dn2))/(iter(i)));
    bS(i,4) = double(((3*Dp1)-(3*Dn1))/(iter(i)));
    bS(i,5) = double(((3*Dp2)-(3*D))/(iter(i)));
    bS(i,6) = double(((3*Dp3)-(3*Dp1))/(iter(i)));
    bS(i,7) = double((5*Dp3)/(2*iter(i)) - (2*Dp2)/(iter(i)) - (Dp1)/
(2*iter(i)));
```

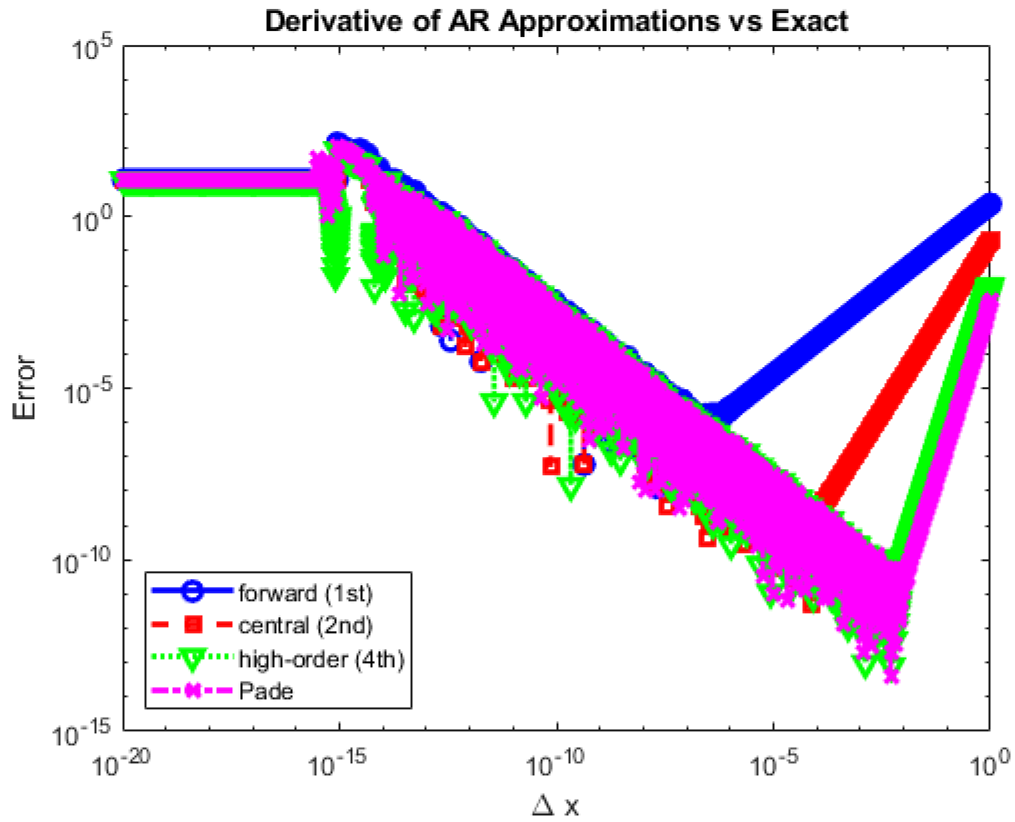
```
    fpAR(i,:) = A\bAR(i,:)';
    fpS(i,:) = A\bS(i,:)';
end

meth = 'fof';
dDdARFoF = dfxAR(10,constants,meth);
dDdSFoF = dfxS(25,constants,meth);
meth = 'soc';
dDdARSoC = dfxAR(10,constants,meth);
dDdSSoC = dfxS(25,constants,meth);
meth = 'foc';
dDdARFoC = dfxAR(10,constants,meth);
dDdSFoC = dfxS(25,constants,meth);

%Calculates Exact deriv using sample code
meth = 'complex';
dfAR = dfxAR(10,constants,meth);
dfS = dfxS(25,constants,meth);
%Calculates error of all methods based on exact deriv
errARFoF = abs((dfAR - dDdARFoF));
errARSoC = abs((dfAR - dDdARSoC));
errARFoC = abs((dfAR - dDdARFoC));
errARPade = abs((dfAR - fpAR(:,4)'));
errSFoF = abs((dfS - dDdSFoF));
errSSoC = abs((dfS - dDdSSoC));
errSFoC = abs((dfS - dDdSFoC));
errSPade = abs((dfS - fpS(:,4)'));

%Creates convergence plots
figure(1);
loglog(iter,errARFoF,'-ob','linewidth',2,'markersize',7);
hold on;
loglog(iter,errARSoC,'--sr','linewidth',2,'markersize',7);
loglog(iter,errARFoC,':vg','linewidth',2,'markersize',7);
loglog(iter,errARPade,'-.xm','linewidth',2,'markersize',7);
hold off;
set(gca,'fontsize', 20);
xlabel('\Delta x', 'fontsize', 24);
ylabel('Error', 'fontsize', 24);
legend('forward (1st)','central (2nd)','high-order (4th)',...
  'Pade','location','southwest');
set(gca,'fontsize', 10);
title('Derivative of AR Approximations vs Exact')

figure(2);
loglog(iter,errSFoF,'-ob','linewidth',2,'markersize',7);
hold on;
loglog(iter,errSSoC,'--sr','linewidth',2,'markersize',7);
loglog(iter,errSFoC,':vg','linewidth',2,'markersize',7);
loglog(iter,errSPade,'-.xm','linewidth',2,'markersize',7);
hold off;
set(gca,'fontsize', 20);
xlabel('\Delta x', 'fontsize', 24);
```

```
ylabel('Error', 'fontsize', 24);
legend('forward (1st)','central (2nd)','high-order (4th)',...
  'Pade','location','southwest');
set(gca,'fontsize', 10);
title('Derivative of S Approximations vs Exact')

[~,IARFoF] = min(abs(errARFoF));
[~,IARSoC] = min(abs(errARSoC));
[~,IARFoC] = min(abs(errARFoC));
[~,IARPade] = min(abs(errARPade));
[~,ISFoF] = min(abs(errSFoF));
[~,ISSoC] = min(abs(errSSoC));
[~,ISFoC] = min(abs(errSFoC));
[~,ISPade] = min(abs(errSPade));
slopeARFoF = mean(diff(log(errARFoF(1:2704)))./
diff(log(iter(1:2704))));
slopeARSoC = mean(diff(log(errARSoC(1:1676)))./
diff(log(iter(1:1676))));
slopeARFoC = mean(diff(log(errARFoC(1:789)))./diff(log(iter(1:789))));
slopeARPade = mean(diff(log(errARPade(1:728)))./
diff(log(iter(1:728))));
slopeSFoF = mean(diff(log(errSFoF(1:2704)))./diff(log(iter(1:2704))));
slopeSSoC = mean(diff(log(errSSoC(1:1676)))./diff(log(iter(1:1676))));
slopeSFoC = mean(diff(log(errSFoC(1:789)))./diff(log(iter(1:789))));
slopeSPade = mean(diff(log(errSPade(1:728)))./diff(log(iter(1:728))));

txt1 = 'The best delta x for dD/dAR using First Order Forward is
 %.1e \nand the derivative at that point is %.15f \nand the slope of
 convergence is %f\n\n';
txt2 = 'The best delta x for dD/dAR using Second Order Central is
 %.1e \nand the derivative at that point is %.15f \nand the slope of
 convergence is %f\n\n';
txt3 = 'The best delta x for dD/dAR using Fourth Order Central is
 %.1e \nand the derivative at that point is %.15f \nand the slope of
 convergence is %f\n\n';
txt4 = 'The best delta x for dD/dAR using Pade is %.1e \nand the
 derivative at that point is %.15f \nand the slope of convergence is
 %f\n\n';
txt5 = 'The best delta x for dD/dS using First Order Forward is
 %.1e \nand the derivative at that point is %.15f \nand the slope of
 convergence is %f\n\n';
txt6 = 'The best delta x for dD/dS using Second Order Central is
 %.1e \nand the derivative at that point is %.15f \nand the slope of
 convergence is %f\n\n';
txt7 = 'The best delta x for dD/dS using Fourth Order Central is
 %.1e \nand the derivative at that point is %.15f \nand the slope of
 convergence is %f\n\n';
txt8 = 'The best delta x for dD/dS using Pade is %.1e \nand the
 derivative at that point is %.15f \nand the slope of convergence is
 %f\n\n\n';
txt = append(txt1,txt2,txt3,txt4,txt5,txt6,txt7,txt8);
fprintf(txt,iter(IARFoF),dDdARFoF(IARFoF),slopeARFoF,iter(IARSoC),dDdARSoC(IARSoC)

 iter(IARPade),fpAR(IARPade,4),slopeARPade,iter(ISFoF),dDdSFoF(ISFoF),slopeSFoF,it
```

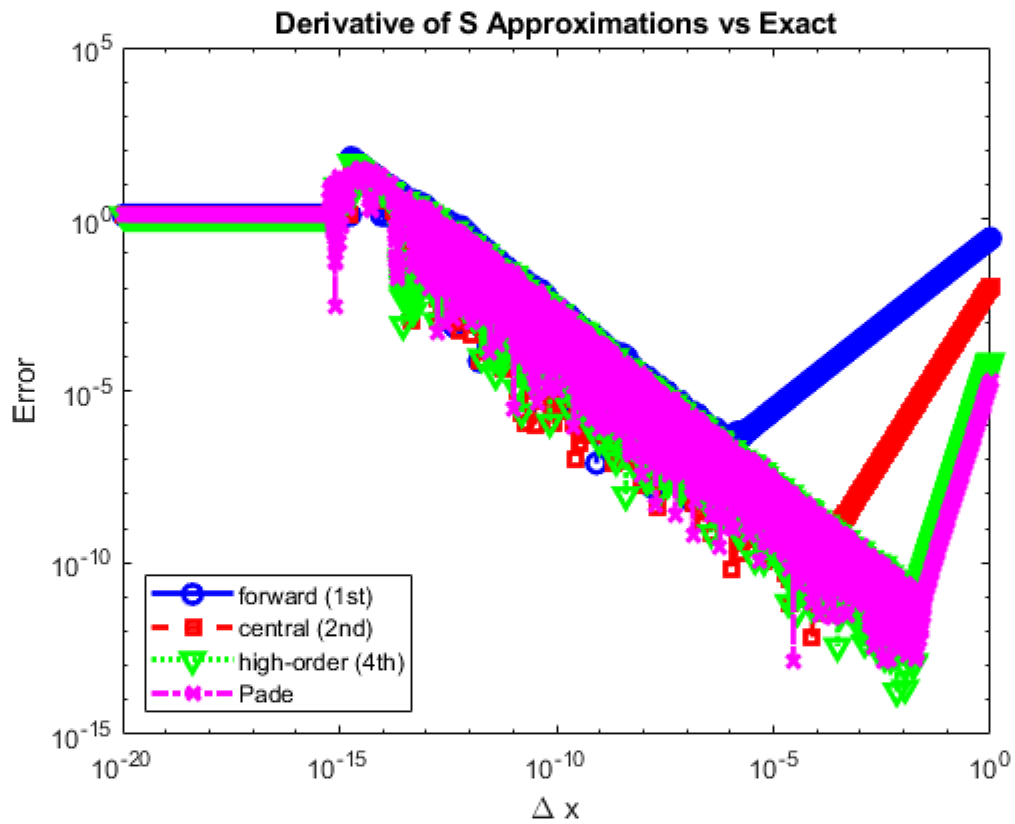
```
    slopeSFoC,iter(ISPade),fpS(ISPade,4),slopeSPade);

% 2.2.1.4
options = optimoptions('fmincon','Display','off','TolX',1e-6,...
  'TolFun', 1e-12,'TolCon', 1e-12,'MaxIter',1e3,'MaxFunEvals',1e4,...
  'Algorithm','sqp');
lb = [0.01,0.01]; %Lower bounds of AR and S
ub = [50,100]; %Upper bounds of AR and S, set larger than they should
 be
AR0 = (lb(1) + ub(1))/2; %initial guess between values of lb and ub
S0 = (lb(2) + ub(2))/2;

% Find optimal values of AR and S given constraints with no gradient
xNoGrad = fmincon(@(x) dragOpt(x,constants), [AR0,S0],[],[],[],[],lb,
[],...
    @(x) constraint(x,constants),options);

%Changes options to include gradient
options =
 optimoptions('fmincon','SpecifyObjectiveGradient',true,'Display',...
    'off','TolX',1e-6,'TolFun', 1e-12,'TolCon',
 1e-12,'MaxIter',1e3,'MaxFunEvals',...
    1e4,'Algorithm','sqp');

% Find optimal values of AR and S given constraints with gradient in
 FoF
xFoF = fmincon(@(x)
 dragOpt(x,constants,dDdARFoF(IARFoF),dDdSFoF(ISFoF)),...
    [AR0,S0],[],[],[],[],lb,[],@(x) constraint(x,constants),options);

% Find optimal values of AR and S given constraints with gradient in
 SoC
xSoC = fmincon(@(x)
 dragOpt(x,constants,dDdARSoC(IARSoC),dDdSSoC(ISSoC)),...
    [AR0,S0],[],[],[],[],lb,[],@(x) constraint(x,constants),options);

% Find optimal values of AR and S given constraints with gradient in
 FOC
xFoC = fmincon(@(x)
 dragOpt(x,constants,dDdARFoC(IARFoC),dDdSFoC(ISFoC)),...
    [AR0,S0],[],[],[],[],lb,[],@(x) constraint(x,constants),options);

% Find optimal values of AR and S given constraints with gradient in
 Pade
xPade = fmincon(@(x)
 dragOpt(x,constants,fpAR(ISPade,4),fpS(ISPade,4)),...
    [AR0,S0],[],[],[],[],lb,[],@(x) constraint(x,constants),options);

xExact = fmincon(@(x) dragOpt(x,constants,dfAR,dfS),[AR0,S0],[],[],[],
[],...
    lb,[],@(x) constraint(x,constants),options);
```

```
fprintf('The optimal Aspect Ratio is %.15f \nThe optimal Wing Area is
 %.15f \n\n\n',xNoGrad(1),xNoGrad(2))
[Dopt,Clopt,Cdopt] = drag(xNoGrad(1),xNoGrad(2),constants);
LtoDopt = Clopt/Cdopt;
txt1 = 'At the optimal Aspect Ratio and Wing Area: \nThe Drag is
 %.15f';
txt2 = ' \nThe Coefficient of Lift is %.15f \nThe Coefficient of Drag
 is';
txt3 = ' %.15f \nThe Lift to Drag ratio is %.15f\n\n';
txt = append(txt1,txt2,txt3);
fprintf(txt,Dopt,Clopt,Cdopt,LtoDopt);
```

```
The best delta x for dD/dAR using First Order Forward is 1.5e-07
and the derivative at that point is -11.768696404708983
and the slope of convergence is 0.993293

The best delta x for dD/dAR using Second Order Central is 7.2e-05
and the derivative at that point is -11.768696415641957
and the slope of convergence is 2.000856

The best delta x for dD/dAR using Fourth Order Central is 5.3e-03
and the derivative at that point is -11.768696415646623
and the slope of convergence is 4.014452

The best delta x for dD/dAR using Pade is 5.0e-03
and the derivative at that point is -11.768696415646746
and the slope of convergence is 4.135633

The best delta x for dD/dS using First Order Forward is 1.3e-07
and the derivative at that point is -1.299169772874502
and the slope of convergence is 0.992721

The best delta x for dD/dS using Second Order Central is 7.9e-05
and the derivative at that point is -1.299169779074089
and the slope of convergence is 2.000216

The best delta x for dD/dS using Fourth Order Central is 6.9e-03
and the derivative at that point is -1.299169779074760
and the slope of convergence is 3.990694

The best delta x for dD/dS using Pade is 3.4e-03
and the derivative at that point is -1.299169779074645
and the slope of convergence is 4.069851
```



Derivative of AR Approximations vs Exact
Error
Δ x
forward (1st)
central (2nd)
high-order (4th)
Pade



Derivative of S Approximations vs Exact
Error
Δ x
forward (1st)
central (2nd)
high-order (4th)
Pade

Functions

```
function [D,g] = dragOpt(x,constants,dDdAR,dDdS)
AR = x(1);
S = x(2);

k = constants.k; %form factor
e = constants.e; %Oswald Efficiency Factor
a1 = constants.a1;
a2 = constants.a2;
a3 = constants.a3; %empirical coeff in Cd
Nult = constants.Nult;
Wo = constants.Wo;
t2c = constants.t2c;
p = constants.p;
U = constants.U2;
u = constants.u;
q = 0.5*p*(U^2);
b = sqrt(AR*S);
c = b/AR;
Swet = 2.2*S; %wetted SA in m^2

W1 = 1;
t = 0;
    while t == 0
        W2 = ((a1*S) + ((a2 * Nult * (b^3) * sqrt(Wo * (W1 + Wo)))/(S
 * t2c)));
        if abs(W2-W1) < (10^-16)
            t = 1;
        end
        W1 = W2;
    end

Ww = W1;
W = Ww + Wo;

Cl = (W/(q*S));
Re = (p*U*c)/u;
Cf = 0.07./(Re^(1/6));
Cd = (k*Cf*Swet/S) + ((Cl^2)/(pi()*e*AR)) + (a3/S);
D = q*S*Cd;

if nargout > 1
    g = [dDdAR;dDdS];
end

end

function [D,Cl,Cd] = drag(AR,S,constants)
k = constants.k; %form factor
e = constants.e; %Oswald Efficiency Factor
a1 = constants.a1;
a2 = constants.a2;
```

```
a3 = constants.a3; %empirical coeff in Cd
Nult = constants.Nult;
Wo = constants.Wo;
t2c = constants.t2c;
p = constants.p;
U = constants.U2;
u = constants.u;
q = 0.5*p*(U^2);
b = sqrt(AR*S);
c = b/AR;
Swet = 2.2*S; %wetted SA in m^2

W1 = 1;
t = 0;
    while t == 0
        W2 = ((a1*S) + ((a2 * Nult * (b^3) * sqrt(Wo * (W1 + Wo)))/(S
 * t2c)));
        if abs(W2-W1) < (10^-16)
            t = 1;
        end
        W1 = W2;
    end

Ww = W1;
W = Ww + Wo;
Cl = (W/(q*S));
Re = (p*U*c)/u;
Cf = 0.07/(Re^(1/6));
Cd = (k*Cf*Swet)/S + ((Cl^2)/(pi()*e*AR)) + (a3/S);
D = q*S*Cd;

end

function df = dfxS(x0,constants,meth)
param.dx = 1e-308;
delx = (logspace(0,-20,10000));
AR = 10;
switch lower(meth)
  case 'complex'
    % complex step
    x = x0;
    for j = 1:length(x0)
      x(j) = x(j)+1i*param.dx; % write a "1" in front of i to be
 explicit
      [f,~,~] = drag(AR,x,constants);
      df(j) = imag(f)/param.dx;
      x(j) = x0(j);
    end
    case 'fof'
        for j = 1:length(delx)
            df(j) = (drag(AR,x0+delx(j),constants)-
drag(AR,x0,constants))/delx(j);
        end
    case 'soc'
```

```
        for j = 1:length(delx)
            df(j) = (drag(AR,x0+delx(j),constants)-drag(AR,x0-
delx(j),constants))/(2*delx(j));
        end
    case 'foc'
        for j = 1:length(delx)
            df(j) = (drag(AR,x0-(2*delx(j)),constants) -
 (8*drag(AR,x0-delx(j),constants)) +...
                (8*drag(AR,x0+delx(j),constants)) -
 drag(AR,x0+(2*delx(j)),constants))/(12*delx(j));
        end
  otherwise
    error 'unsupported';
end
end

function df = dfxAR(x0,constants,meth)
param.dx = 1e-308;
delx = (logspace(0,-20,10000));
S = 25;
switch lower(meth)
  case 'complex'
    % complex step
    x = x0;
    for j = 1:length(x0)
      x(j) = x(j)+1i*param.dx; % write a "1" in front of i to be
 explicit
      [f,~,~] = drag(x,S,constants);
      df(j) = imag(f)/param.dx;
      x(j) = x0(j);
    end
    case 'fof'
        for j = 1:length(delx)
            df(j) = (drag(x0+delx(j),S,constants)-
drag(x0,S,constants))/delx(j);
        end
    case 'soc'
        for j = 1:length(delx)
            df(j) = (drag(x0+delx(j),S,constants)-drag(x0-
delx(j),S,constants))/(2*delx(j));
        end
    case 'foc'
        for j = 1:length(delx)
            df(j) = (drag(x0-(2*delx(j)),S,constants) - (8*drag(x0-
delx(j),S,constants)) +...
                (8*drag(x0+delx(j),S,constants)) -
 drag(x0+(2*delx(j)),S,constants))/(12*delx(j));
        end
  otherwise
    error 'unsupported';
end
end

function [c,ceq] = constraint(x,constants)
```

```
AR = x(1);
S = x(2);
k = constants.k; %form factor
e = constants.e; %Oswald Efficiency Factor
a1 = constants.a1;
a2 = constants.a2;
a3 = constants.a3; %empirical coeff in Cd
Nult = constants.Nult;
Wo = constants.Wo;
t2c = constants.t2c;
p = constants.p;
U = constants.U2;
u = constants.u;
Uland = constants.Uland;
q = 0.5*p*(Uland^2);
b = sqrt(AR*S);
c = b/AR;
Swet = 2.2*S; %wetted SA in m^2

W1 = 1;
W2 = 2;
t = 0;
    while t == 0
        W2 = ((a1*S) + ((a2 * Nult * (b^3) * sqrt(Wo * (W1 + Wo)))/(S
 * t2c)));
        if abs(W2-W1) < (10^-16)
            t = 1;
        end
        W1 = W2;
    end
Ww = W1;
W = Ww + Wo;


c = [(W)/(0.5*p*((constants.Uland)^2)*constants.Clmax)-
x(2),sqrt(x(1)*x(2))-20];
ceq = [];
end
```

```
The optimal Aspect Ratio is 13.439578163583803
The optimal Wing Area is 23.950303381668459


At the optimal Aspect Ratio and Wing Area:
The Drag is 499.337841137262387
The Coefficient of Lift is 1.200460927781807
The Coefficient of Drag is 0.055597106856910
The Lift to Drag ratio is 21.592147427229044
```

Published with MATLAB® R2021a