16.90 Project 1

Spring 2014

1 Problem 1

Introduction

This project examines an aeroelastic airfoil. This can be imagined as a section with a torsional and linear spring, as given in the problem statement. The governing equations are given by

$$M_{hh}\frac{d^2h}{dt^2} + M_{h\alpha}\frac{d^2\alpha}{dt^2} + D_h\frac{dh}{dt} + K_h h + L = 0 \quad (1)$$

$$M_{\alpha\alpha}\frac{d^2\alpha}{dt^2} + M_{\alpha h}\frac{d^2h}{dt^2} + D_\alpha\frac{d\alpha}{dt} + K_\alpha(1 + k_{NL}h^2)\alpha + M = 0 \quad (2)$$

The parameters are given by

$$M_{hh} = 1 \qquad M_{h\alpha} = 0.625 \qquad M_{\alpha\alpha} = 1.25$$
$$M_{\alpha h} = 0.25 \qquad D_h = 0.1s^{-1} \qquad D_\alpha = 0.25s^{-1}$$
$$K_h = 0.2s^{-2} \qquad K_\alpha = 1.25s^{-2} \qquad k_{NL} = 10$$

Additionally, the lift L and the moment M can be expressed as functions of the dynamic pressure Q. $Q = 1$ at the design airspeed V_{NO}.

$$L = 1s^{-2}Q\alpha \quad (3)$$

$$M = -0.7s^{-2}Q\alpha \quad (4)$$

The initial conditions at $t = 0$ are given by

$$h(0) = 0$$
$$\frac{dh}{dt}(0) = 0$$
$$\frac{d\alpha}{dt}(0) = 0$$
$$0 < \alpha(0) < 0.08 \text{ rad}$$

1.1

We now solve the equations of motion given by (1) and (2) for 60 seconds using three different schemes. The unknowns for each time step are α, h, $\frac{d\alpha}{dt}$, and $\frac{dh}{dt}$.

1.1.1 Forward Euler

The forward Euler scheme is given by

$$v^{n+1} = v^n + f(v^n)\Delta t$$

Thus, for the four variables, we have the following discretization

$$\begin{aligned}
\alpha^{n+1} &= \alpha^n + \frac{d\alpha^n}{dt}\Delta t \\
h^{n+1} &= h^n + \frac{dh^n}{dt}\Delta t \\
\frac{d\alpha}{dt}^{n+1} &= \frac{d\alpha^n}{dt} + \frac{d^2\alpha^n}{dt^2}\Delta t \\
\frac{dh}{dt}^{n+1} &= \frac{dh^n}{dt} + \frac{d^2h^n}{dt^2}\Delta t
\end{aligned}$$

1.1.2 Midpoint

The midpoint scheme is given by

$$v^{n+1} = v^{n-1} + 2\Delta t f(v^n)$$

This gives the following discretization.

$$\begin{aligned}
\alpha^{n+1} &= \alpha^{n-1} + 2\frac{d\alpha}{dt}^n\Delta t \\
h^{n+1} &= h^{n-1} + 2\frac{dh}{dt}^n\Delta t \\
\frac{d\alpha}{dt}^{n+1} &= \frac{d\alpha}{dt}^{n-1} + 2\frac{d^2\alpha}{dt^2}^n\Delta t \\
\frac{dh}{dt}^{n+1} &= \frac{dh}{dt}^{n-1} + 2\frac{d^2h}{dt^2}^n\Delta t
\end{aligned}$$

1.1.3 Backwards Differentiation BDF-2

The backwards differentiation method is given by

$$v^{n+1} + \sum_{i=1}^{s} \alpha_i v^{n+1-i} = \Delta t \beta_0 f^{n+1}$$

For the second order scheme this works out to

$$v^{n+1} = \frac{4}{3}v^n - \frac{1}{3}v^{n-1} + \frac{2}{3}\Delta t f(v^{n+1})$$

$$\alpha^{n+1} = \frac{4}{3}\alpha^n - \frac{1}{3}\alpha^{n-1} + \frac{2}{3}\Delta t \frac{d\alpha}{dt}^{n+1}$$
$$h^{n+1} = \frac{4}{3}h^n - \frac{1}{3}h^{n-1} + \frac{2}{3}\Delta t \frac{dh}{dt}^{n+1}$$
$$\frac{d\alpha}{dt}^{n+1} = \frac{4}{3}\frac{d\alpha}{dt}^n - \frac{1}{3}\frac{d\alpha}{dt}^{n-1} + \frac{2}{3}\Delta t \frac{d^2\alpha}{dt^2}^{n+1}$$
$$\frac{dh}{dt}^{n+1} = \frac{4}{3}\frac{dh}{dt}^n - \frac{1}{3}\frac{dh}{dt}^{n-1} + \frac{2}{3}\Delta t \frac{d^2h}{dt^2}^{n+1}$$

1.1.4 Solution Method

Equations (1) and (2) are solved for the second derivatives $\frac{d^2h}{dt^2}$ and $\frac{d^2\alpha}{dt^2}$. We also substitute in the equations for L and M given by (3) and (4) respectively. This reduces to

$$\frac{d^2h}{dt^2} = \frac{\frac{M_{h\alpha}}{M_{\alpha\alpha}}\left(D_\alpha \frac{d\alpha}{dt} + K_\alpha(1 + k_{NL}h^2)\alpha - 0.7Q\alpha\right) - D_h\frac{dh}{dt} - K_h h - Q\alpha}{M_{hh} - \frac{M_{\alpha h}M_{h\alpha}}{M_{\alpha\alpha}}} \tag{5}$$
$$\frac{d^2\alpha}{d^2} = \frac{\frac{M_{\alpha h}}{M_{hh}}\left(D_h \frac{dh}{dt} + K_h h + Q\alpha\right) - D_\alpha\frac{d\alpha}{dt} - K_\alpha\left(1 + k_{NL}h^2\right)\alpha + 0.7Q\alpha}{M_{\alpha\alpha} - \frac{M_{\alpha h}M_{h\alpha}}{M_{hh}}} \tag{6}$$

For the forward Euler and mid-point schemes, the code implementation is simple. We simply code the scheme directly into MATLAB®ÈSince we know the initial state, we use the initial values to calculate the second time step and proceed from there. For the BDF-2 scheme, we must use a Newton-Raphson method to solve the implicit system since the scheme requires the derivative of the next (unkonwn) timestep.

The Newton-Raphson residual is calculated as follows.

$$v^{n+} = \frac{4}{3}v^n - \frac{1}{3}v^{n-1} + \frac{2}{3}\Delta t f(v^{n+1})$$
$$R(w) = w - \frac{4}{3}v^n + \frac{1}{3}v^{n-1} - \frac{2}{3}\Delta t f(w, t^{n+1})$$

Next, we derive the state update Δw.

$$\begin{aligned}
&w^{m+1} = w^m + \Delta w \\
&R(w^{m+1}) = 0 \\
&R(w^m + \Delta w) = 0 \\
&R(w^m) + \left.\frac{\partial R}{\partial w}\right|_{w^m} \Delta w = 0 \\
&\left.\frac{\partial R}{\partial w}\right|_{w^m} \Delta w = -R(w^m) \\
&\frac{\partial R}{\partial w} = I - \frac{2}{3}\Delta t \left.\frac{\partial f}{\partial w}\right|_{w^m} \\
&\Rightarrow \Delta w = -\left[\left.\frac{\partial R}{\partial w}\right|_{w^m}\right]^{-1} R(w^m)
\end{aligned}$$

The derivative of the residual $\frac{\partial R}{\partial w}$ is given by the Jacobian of the residual. The Newton-Raphson process is

1. Compute the Jacobian
2. Compute the change in the state Δw
3. Compute the new state w
4. Compute the residual
5. Check if the residual is "small enough." The tolerance will be determined later in this project
6. If the residual is not small enough, go back to step 1 and recompute using the new state

First, we must find the Jacobian. Note that $(\dot{\ })$ denotes a time derivative and $(\ddot{\ })$ denotes the second time derivative.

$$f = \begin{bmatrix} \dot{\alpha} \\ \ddot{\alpha} \\ \dot{h} \\ \ddot{h} \end{bmatrix}$$

$$w = \begin{bmatrix} \alpha \\ \dot{\alpha} \\ h \\ \dot{h} \end{bmatrix}$$

$$\Rightarrow \frac{\partial f}{\partial w} = \begin{bmatrix} \frac{\partial \dot{\alpha}}{\partial \alpha} & \frac{\partial \dot{\alpha}}{\partial \dot{\alpha}} & \frac{\partial \dot{\alpha}}{\partial h} & \frac{\partial \dot{\alpha}}{\partial \dot{h}} \\ \frac{\partial \ddot{\alpha}}{\partial \alpha} & \frac{\partial \ddot{\alpha}}{\partial \dot{\alpha}} & \frac{\partial \ddot{\alpha}}{\partial h} & \frac{\partial \ddot{\alpha}}{\partial \dot{h}} \\ \frac{\partial \dot{h}}{\partial \alpha} & \frac{\partial \dot{h}}{\partial \dot{\alpha}} & \frac{\partial \dot{h}}{\partial h} & \frac{\partial \dot{h}}{\partial \dot{h}} \\ \frac{\partial \ddot{h}}{\partial \alpha} & \frac{\partial \ddot{h}}{\partial \dot{\alpha}} & \frac{\partial \ddot{h}}{\partial h} & \frac{\partial \ddot{h}}{\partial \dot{h}} \end{bmatrix}$$

These terms are all derived below.

$$\dot{\alpha} = \dot{\alpha}$$

$$\ddot{\alpha} = \frac{\frac{M_{\alpha h}}{M_{hh}}\left(D_h \frac{dh}{dt} + K_h h + Q\alpha\right) - D_\alpha \frac{d\alpha}{dt} - K_\alpha \left(1 + k_{NL} h^2\right) \alpha + 0.7Q\alpha}{M_{\alpha\alpha} - \frac{M_{\alpha h} M_{h\alpha}}{M_{hh}}}$$

$$\dot{h} = \dot{h}$$

$$\ddot{h} = \frac{\frac{M_{h\alpha}}{M_{\alpha\alpha}}\left(D_\alpha \frac{d\alpha}{dt} + K_\alpha (1 + k_{NL} h^2)\alpha - 0.7Q\alpha\right) - D_h \frac{dh}{dt} - K_h h - Q\alpha}{M_{hh} - \frac{M_{\alpha h} M_{h\alpha}}{M_{\alpha\alpha}}}$$

Computing the Jacobian terms

$$\frac{\partial \dot{\alpha}}{\partial \alpha} = 0 \quad \frac{\partial \dot{\alpha}}{\partial \dot{\alpha}} = 1 \quad \frac{\partial \dot{\alpha}}{\partial h} = 0 \quad \frac{\partial \dot{\alpha}}{\partial \dot{h}} = 0$$

$$\frac{\partial \ddot{\alpha}}{\partial \alpha} = \frac{\frac{M_{\alpha h}}{M_{hh}} Q - K_\alpha \left(1 + k_{NL} h^2\right) + 0.7Q}{M_{\alpha\alpha} - \frac{M_{\alpha h} M_{h\alpha}}{M_{hh}}}$$

$$\frac{\partial \ddot{\alpha}}{\partial \dot{\alpha}} = \frac{-D_\alpha}{M_{\alpha\alpha} - \frac{M_{\alpha h} M_{h\alpha}}{M_{hh}}}$$

$$\frac{\partial \ddot{\alpha}}{\partial h} = \frac{\frac{M_{\alpha h}}{M_{hh}} K_h - 2K_\alpha k_{NL} h\alpha}{M_{\alpha\alpha} - \frac{M_{\alpha h} M_{h\alpha}}{M_{hh}}}$$

$$\frac{\partial \ddot{\alpha}}{\partial \dot{h}} = \frac{\frac{M_{\alpha h}}{M_{hh}} D_h}{M_{\alpha\alpha} - \frac{M_{\alpha h} M_{h\alpha}}{M_{hh}}}$$

$$\frac{\partial \dot{h}}{\partial \alpha} = 0 \quad \frac{\partial \dot{h}}{\partial \dot{\alpha}} = 0 \quad \frac{\partial \dot{h}}{\partial h} = 0 \quad \frac{\partial \dot{h}}{\partial \dot{h}} = 1$$

$$\frac{\partial \ddot{h}}{\partial \alpha} = \frac{\frac{M_{h\alpha}}{M_{\alpha\alpha}} \left(K_\alpha (1 + k_{NL} h^2) - 0.7Q\right) - Q}{M_{hh} - \frac{M_{\alpha h} M_{h\alpha}}{M_{\alpha\alpha}}}$$

$$\frac{\partial \ddot{h}}{\partial \dot{\alpha}} = \frac{\frac{M_{h\alpha}}{M_{\alpha\alpha}} D_\alpha}{M_{hh} - \frac{M_{\alpha h} M_{h\alpha}}{M_{\alpha\alpha}}}$$

$$\frac{\partial \ddot{h}}{\partial h} = \frac{\frac{M_{h\alpha}}{M_{\alpha\alpha}} \left(2K_\alpha k_{NL} h\alpha\right) - K_h}{M_{hh} - \frac{M_{\alpha h} M_{h\alpha}}{M_{\alpha\alpha}}}$$

$$\frac{\partial \ddot{h}}{\partial \dot{h}} = \frac{-D_h}{M_{hh} - \frac{M_{\alpha h} M_{h\alpha}}{M_{\alpha\alpha}}}$$

This formulation allows the Jacobian to be coded into a MATLAB function that can be easily computed for each state.

For computation, two functions to compute the state and the Jacobian were written. The function **Aerovibe.m** takes in the state and computes the derivatives of the state using

equations (5) and (6) for the second derivatives of α and h and $\frac{d\alpha}{dt} = \dot{\alpha}$ and $\frac{dh}{dt} = \dot{h}$ for the first derivatives. Here, $\dot{\alpha}$ and $\dot{h}$ are the current state derivatives. **AeroVibeJacobian.m** uses the equations listed on page 6 to compute the Jacobian of the aeroelastic wing model. See the appendix A for the printout of the code for these two functions.

To compute the schemes, a general function was written for each one. These are attached in the appendix as well.

To compute the forward Euler solution, the function **ForwardEuler.m** takes in the function name of the ODE being examined, the timespan vector, the initial state, and the time step Δt. It then uses a for loop to compute the forward Euler solution. The ODE function computes the derivatives of the state.

To compute the mid-point solution, a function similar to the forward Euler one is used. **Midpoint.m** takes in the ODE computation function, the initial state, the time span, and the time step size. The midpoint function computes the v^2 state using a forward Euler scheme and then computes all subsequent v^n using the midpoint scheme.

For the Backward Difference scheme, the function **BDF2.m** takes in the ODE function name, the Jacobian function name, the initial state, the time span, the timestep size dt, and the tolerance. As outlined on page 4, **BDF2.m** uses a Newton-Raphson scheme to solve the implicit function. First, v^2 is computed using the forward Euler scheme. Then, for each time step, the process from page 4 is used. Once the desired tolerance is reached v^{n+1} is set to w and the code moves on to the next time step. The Jacobian $\left.\frac{\partial R}{\partial f}\right|_w$ is computed by using the Jacobian function used in the input to **BDF.m**, and the derivative is computed by the ODE function.

To summarize, the inputs and outputs for each scheme are listed below in pseudo-code.

$$[t, x] = \text{ForwardEuler('AeroVibe'},\ x_0,\ [t_{min}\ t_{max}],\ dt)$$
$$[t, x] = \text{Midpoint('AeroVibe'},\ x_0,\ [t_{min}\ t_{max}],\ dt)$$
$$[t, x] = \text{BDF2('AeroVibe', 'AeroVibeJacobian'},\ x_0,\ [t_{min}\ t_{max}],\ dt,\ tol)$$

where

$$x_0 = \begin{bmatrix} \alpha_0 \\ \dot{\alpha}_0 \\ h_0 \\ \dot{h}_0 \end{bmatrix}$$

and *tol* is the tolerance of the Newton-Raphson iterative solution. The outputs, $[t, x]$ are the time vector and matrix of the solutions for each time step, respectively.

1.1.5 Sensitivity Analysis

Before we can analyze the system, we must first determine the tradeoff between the time step size and the accuracy of the solution. In order to analyze this system, we must first

pick a baseline with which we can compare various values of dt and their solutions. In order to do this, we choose to use the BDF-2 scheme. Since it is implicit and this is a non-linear system, we assume that it will be the most accurate. We now set about proving it.

First, we pick a "very small" value for Δt and the tolerance of the BDF-2 Newton iteration loop. After a small amount of trial and error, we pick $\Delta t = 10^{-6}$ and a tolerance of 10^{-8}. Both of these values are quite small and result in a 1+ hour computation time. We then save the values of the output for $Q = 1$ and $Q = 1.5$ in order to compare the other schemes with them.

To do the sensitivity analysis, we run a series of trials with each scheme for an array of values of Δt. These values of Δt are all an order of magnitude different from each other. For each timestep size and each value of Q we compute the solution and find the maximum error. This is done by subtracting the most accurate solution from the scheme solution and taking the absolute value.

Succinctly the process is as follows:

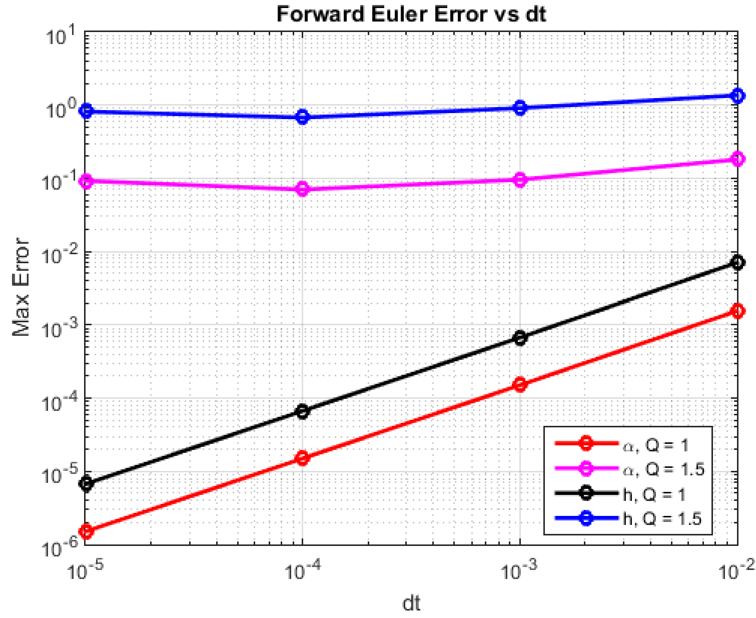
1. Choose scheme
2. Choose Δt
3. Compute solution for $Q = 1$ and $Q = 1.5$
4. Find the maximum, $|\text{scheme solution} - \text{most accurate}|$
5. Save the maximum error for h and α and plot the results

For each analysis, the dt vector was

$$dt = [10^{-2}\ 10^{-3}\ 10^{-4}\ 10^{-5}]$$

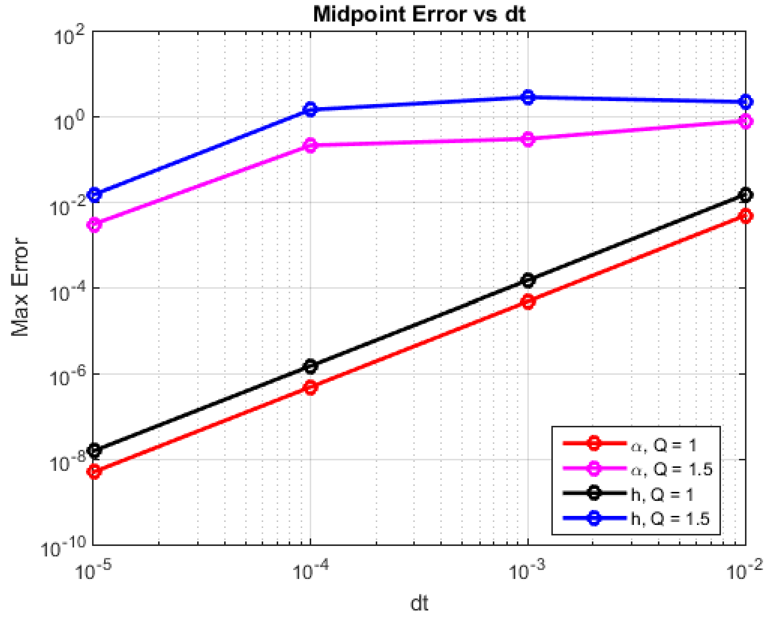
The initial α_0 was 0.08.

The results show the accuracy of each scheme. By examining the slope of the dt versus error lines, we can find the order of accuracy. The results for the forward Euler scheme sensitivity analysis are shown below.



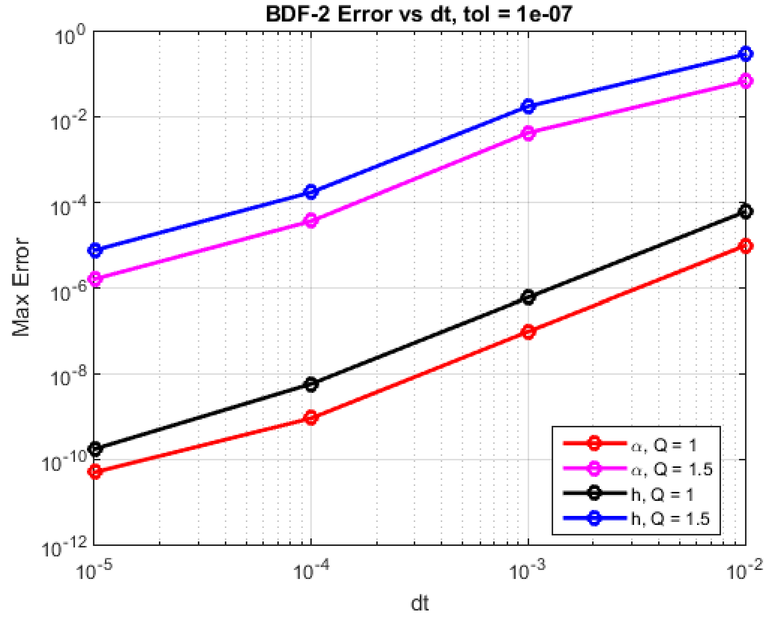
The slope of the forward Euler Δt versus error plot is 1. This means that the forward Euler scheme is first order accurate. Thus, as the order of Δt decreases by 1, so does the order of the error. Thus, for a 10 fold refinement of the time step size, the error also decreases by a factor of 10. Interestingly, at $Q = 1.5$ the error appears to be constant no matter the change in the time step. This most likely indicates that the system itself falls apart near $Q = 1.5$. The governing equations probably move toward a singularity near this value of Q. The aeroelastic equations of motion examined in this project are linear approximations of the true behavior of the system. The most likely reason that the $Q = 1.5$ solution does not improve for a refined temporal grid is that the equations of motion that we are using become invalid at this speed. Overall, the forward Euler method is not extremely efficient, since it is only first order accurate.

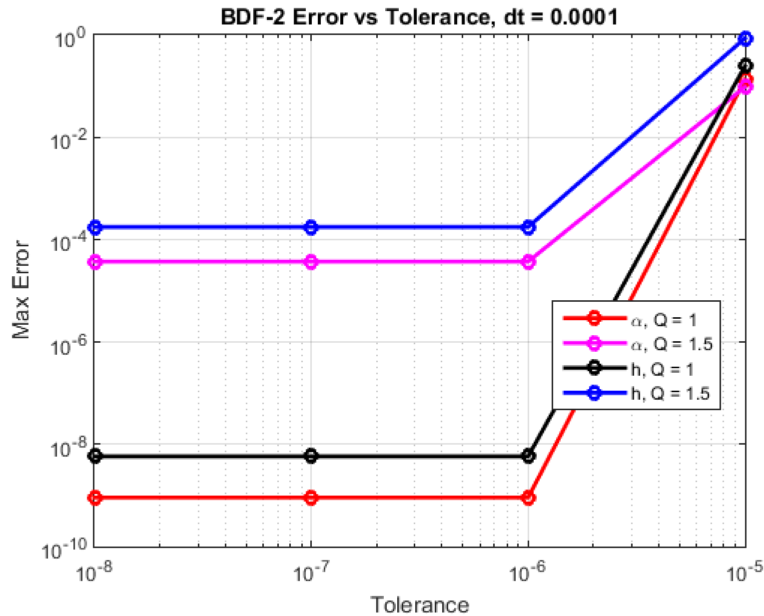
The midpoint scheme sensitivity is shown below.



For the midpoint scheme, the sensitivity analysis shows that for $Q = 1$ the height and α solutions are 2nd order accurate. The slope of the error versus Δt line is 2. This means that the scheme is more accurate than the forward Euler scheme as $\Delta t \to 0$. In addition, we see that as Δt goes to 0, the error of the $Q = 1.5$ solution begins to decrease. One issue, however, is the size of the error for $Q = 1.5$. The results show that the error is greater than $10^0 = 1$ for $\Delta t = 10^{-4}$. This means that although the trend is good, i.e. error decreases as Δt goes to 0, the absolute error is very large. Thus, although the 2nd order accuracy is attractive, the sensitivity analysis shows that the scheme is unstable as Δt goes to ∞. Thus, despite the 2nd order accuracy being efficient, the results for $Q = 1.5$ are probably suspect and must be examined further.

For the BDF-2 scheme, two different sensitivity analyses were performed: one for the time step size Δt and one for the Newton iteration tolerance. The results are plotted below.





The results show that the scheme is 2nd order accurate for both $Q = 1$ and $Q = 1.5$ with respect to the time step size Δt. This means that our initial guess of the BDF-2 scheme being the most efficient was correct. For the Δt analysis, a tolerance of 10^{-7} was used to speed up the computations while still guaranteeing accuracy. This choice is validated by the results of the tolerance sensitivity analysis of the BDF-2 scheme. This analysis was the same as the Δt analysis except that a vector of tolerances was analyzed. This vector was

$$tol = [10^{-5}\ 10^{-6}\ 10^{-7}\ 10^{-8}]$$

The time step was $\Delta t = 10^{-4}$. The results show that for a tolerance greater than 10^{-6} the error was very high, around 10^{0}. However, for tolerance values of 10^{-6} or smaller, the error

was constant. These results show that for the BDF-2 scheme, there is a tolerance for which the scheme becomes most accurate for a specific value of Δt. If the tolerance is decreased, there is no benefit in the error. The error remains constant but computation time increases substantially for smaller values of the tolerance. Thus, this is an efficient scheme. We obtain 2 order of accuracy for 1 order of loss in efficiency (Δt).

Using the results of the sensitivity analysis, the following parameters were used to calculate the solution.

scheme	Δt	tolerance	max error from sensitivity analysis	
			$Q = 1.0$	$Q = 1.5$
Forward Euler	$1e-4$	N/A	10^{-4}	10^{0}
Midpoint	$1e-5$	N/A	10^{-8}	10^{-2}
BDF-2	$1e-4$	$1e-6$	10^{-8}	10^{-4}

These choices of Δt reflect the values of Δt which resulted in the least error for $Q = 1.5$ while still trying to keep the code running quickly. For the forward Euler scheme, the minimum error was obtained when $\Delta t = 10^{-4}$. Therefore, since computation time for this time step size was tolerable, this choice was the logical one. For this choice of Δt we expect the following error.

	$Q = 1$	$Q = 1.5$
α	1.05×10^{-5}	1.6×10^{-2}
h	1.6×10^{-5}	1.5×10^{-1}

For the midpoint scheme, the choice of Δt again is limited by the error by the $Q = 1.5$ case. The sensitivity analysis showed that the error for $Q = 1.5$ was above 10^{-1} for all $\Delta t \geq 10^{-4}$. Therefore, the only way to obtain an accurate solution is to reduce the time step size. The tradeoff here is the computation time. A Δt of 10^{-5} takes 10 times as many computations as a Δt of 10^{-4}. Therefore, we get an accuracy that we desire but at a low efficiency. We expect the following errors for $\Delta t = 10^{-5}$ for the midpoint scheme.

	$Q = 1$	$Q = 1.5$
α	1.4×10^{-9}	1.2×10^{-3}
h	1.0×10^{-8}	1.0×10^{-2}

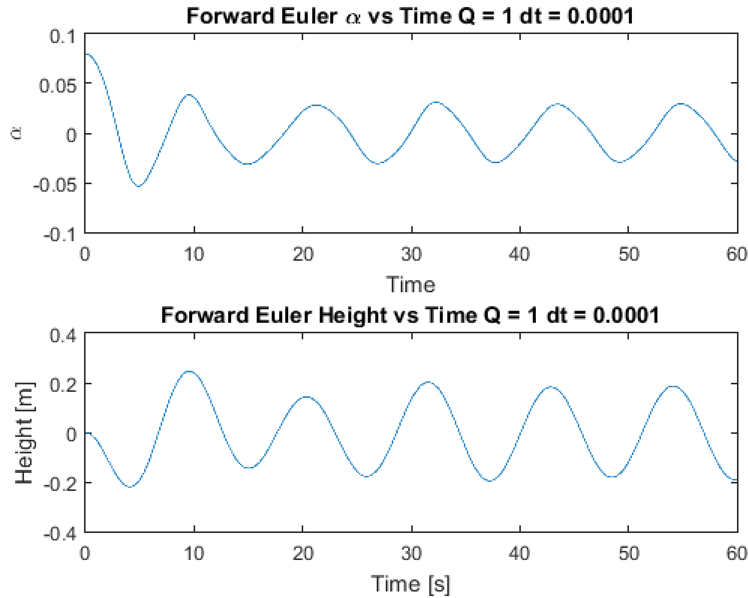
For the BDF-2 scheme, the choice of Δt is much easier. Since a Δt of 10^{-3} already has a maximum error of 10^{-2} for $Q = 1.5$, which is the same as the error for the midpoint scheme at $\Delta t = 10^{-5}$, the BDF-2 choice is much easier. The same is true for the tolerance value. The analysis showed that for $\Delta t = 10^{-4}$ a tolerance of 10^{-6} or smaller results in the same

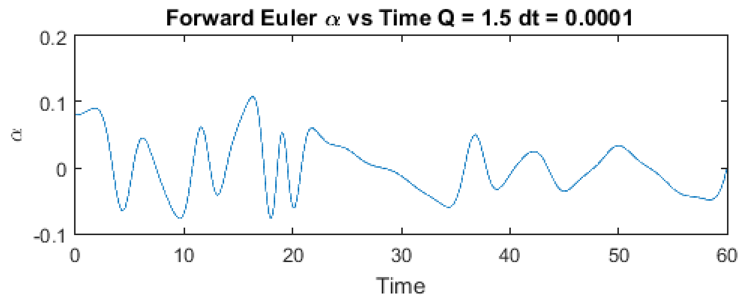
error. Thus, the choice of $\Delta t = 10^{-4}$ and tolerance of 10^{-6} are both very good choices for evaluating this system. In addition, these choices both have a tolerable computation time, making them ideal choices for this particular system.

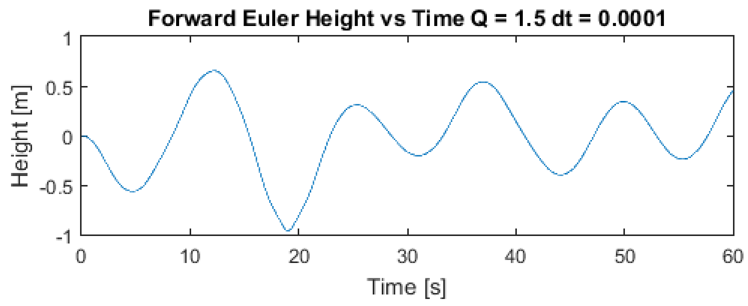
For a $\Delta t = 10^{-4}$ and a tolerance of 10^{-6} we expect the following maximum errors.

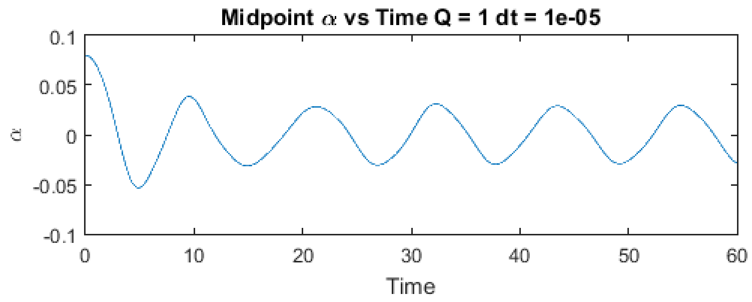
	$Q = 1$	$Q = 1.5$
α	1.0×10^{-9}	1.5×10^{-5}
h	1.7×10^{-9}	1.1×10^{-4}

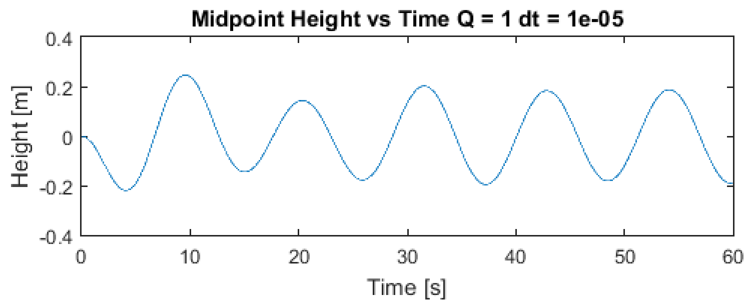
Once the sensitivity analysis was completed, the numerical solutions for each scheme were found. Using the values in the above table for Δt, the numerical results for each scheme were computed and plotted for $0 \leq t \leq 60$ and $Q = 1$ and $Q = 1.5$. The plots of these solutions are below.

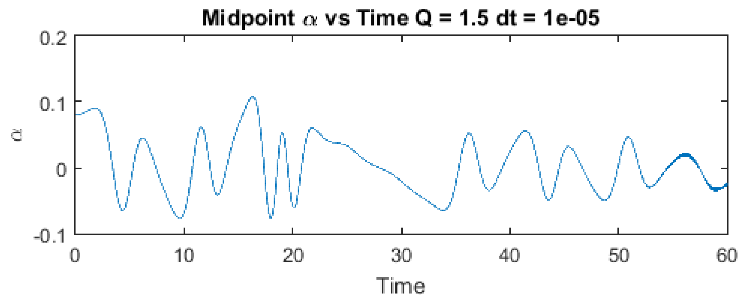


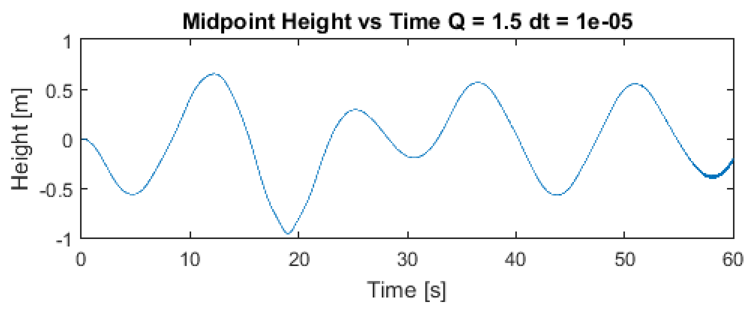

Forward Euler α vs Time Q = 1.5 dt = 0.0001
α
Time

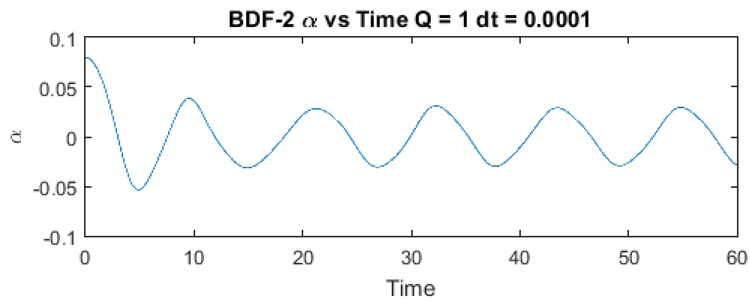

Forward Euler Height vs Time Q = 1.5 dt = 0.0001
Height [m]
Time [s]

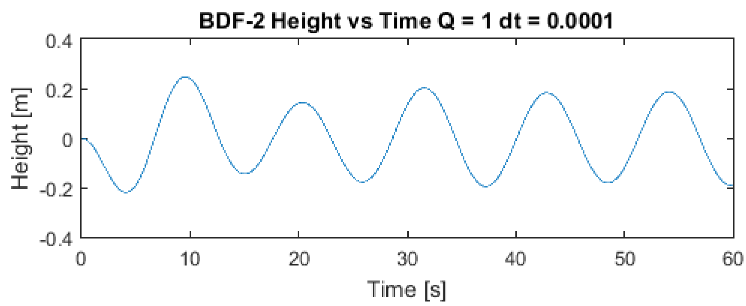

Midpoint α vs Time Q = 1 dt = 1e-05
α
Time

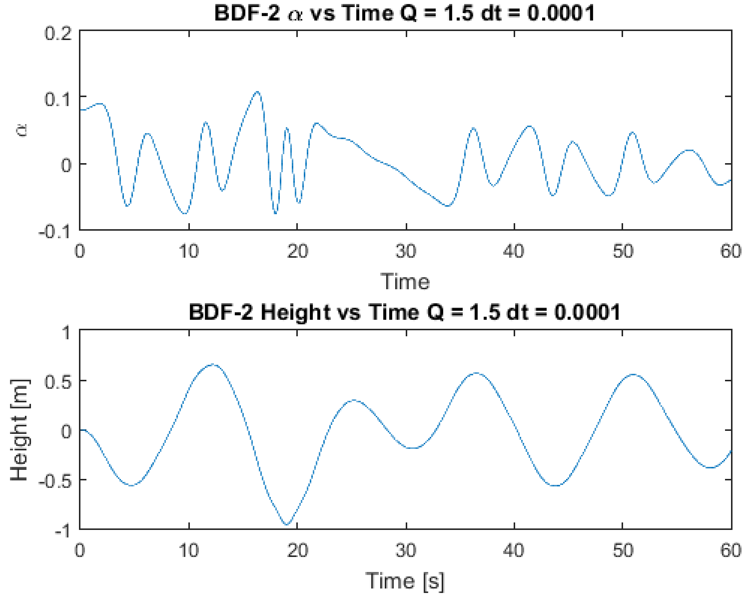

Midpoint Height vs Time Q = 1 dt = 1e-05
Height [m]
Time [s]


Midpoint α vs Time Q = 1.5 dt = 1e-05
α
Time


Midpoint Height vs Time Q = 1.5 dt = 1e-05
Height [m]
Time [s]


BDF-2 α vs Time Q = 1 dt = 0.0001
α
Time


BDF-2 Height vs Time Q = 1 dt = 0.0001
Height [m]
Time [s]



We see from the numerical solutions that for all three schemes for the $Q = 1$ case the solution for both α and h are almost identical. This is as expected since we chose Δt and the tolerance such that the error should be at most 10^{-4} between the three schemes. Therefore all three schemes are good methods for the $Q = 1$ case.

For the $Q = 1.5$ case, the results of the three schemes begin to diverge. The forward Euler and BDF-2 schemes appear to be very close, as we expected. The midpoint scheme, however, gets ”thick” toward the end of the timespan being examined. This shows the instability that was noticed in the sensitivity analysis. The midpoint scheme, even though it is 2nd order accurate and appears to only have a maximum error of 10^{-2} when compared with the most accurate solution under these conditions, is actually globally unstable as t goes to infinity.

These results confirm our initial guess that the BDF-2 scheme is the best one to use for this dynamic system. It is the most accurate at the largest Δt for both speeds. Thus, it solves the system most accurately and most efficiently, a big win-win in computational methods.

With respect to the behavior of the actual system, the results are intriguing. For the $Q = 1$ case, both the height and α plots show periodic oscillations about $\alpha = 0$ and $h = 0$ after about 10 seconds. This means that if the controller fails while the airplane is operating at cruise, the system will oscillate but will remain stable. This is very important because it means that the plane can probably still fly after the controller fails.

When $Q = 1.5$ the results are more worrying with respect to the wing's structural integrity. The numerical models predict that if the controller fails while the airplane is flying at the never-exceed speed, V_{NE}, the wing will twist violently and make a steep swing of almost 1.5 chord lengths. After about 30 seconds the plunging settles into a periodic oscillation about $h = 0$, but the pitch continues to vary in period and amplitude. This means that there is a danger that the wing could pull itself apart if the controller never turns back on. When examined carefully, it appears that the pitching motion is being caused yb the presence of several different modes interacting on the wing. This can be seen when the solution

becomes almost a straight line around $t = 30$, but then quickly becomes oscillatory again approximately 5 seconds later. This behavior suggests presence of a few different eigenmodes in the system. This would also explain the oscillatory nature of the solutions.

In summary, we find through our analysis that the BDF-2 scheme is the most efficient and the most accurate scheme for solving this system.

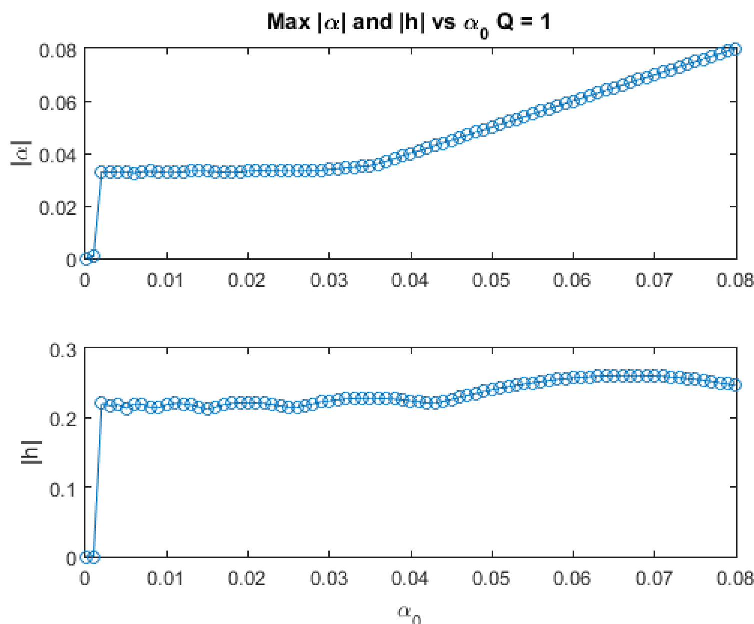
1.2 Maximum Motion

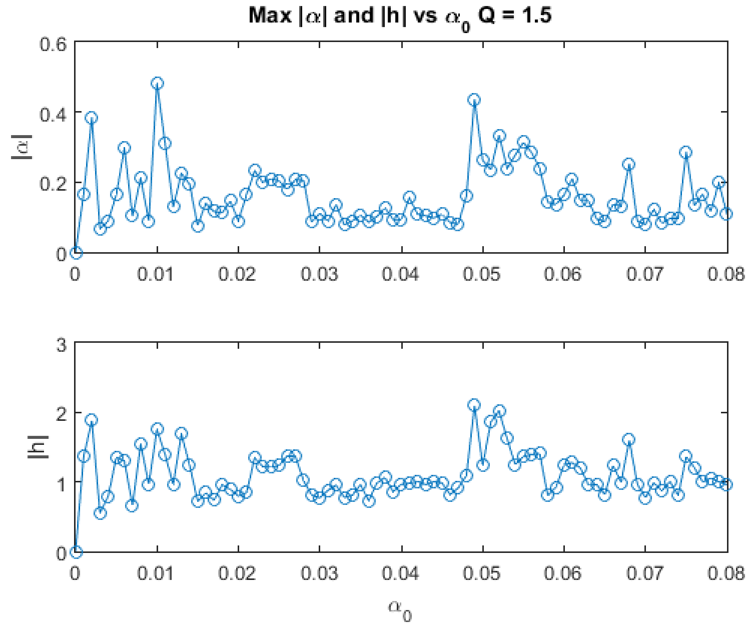
To determine if the plane will fail, we examine a range of initial pitch values from $0 \leq \alpha \leq 0.08$ radians. To evaluate these initial conditions, we use a BDF-2 scheme with $\Delta t = 10^{-3}$ and a tolerance of 10^{-6}. This time step size was chosen based on the results of the sensitivity analysis done in part 1. That analysis showed that for a $\Delta t = 10^{-3}$, the BDF-2 scheme had a maximum deviation from the most accurate scheme of 10^{-2} for the height for $Q = 1.5$. Since the maximum height for $\alpha_0 = 0.08$ was on the order of 1 from part 1, this means that we can expect an error of around 1%, which is a tolerable engineering error. The tolerance of 10^{-6} was most accurate for $\Delta t = 10^{-4}$ as shown in part 1, so it will definitely be accurate enough for a larger time step.

We therefore set up the problem as follows.

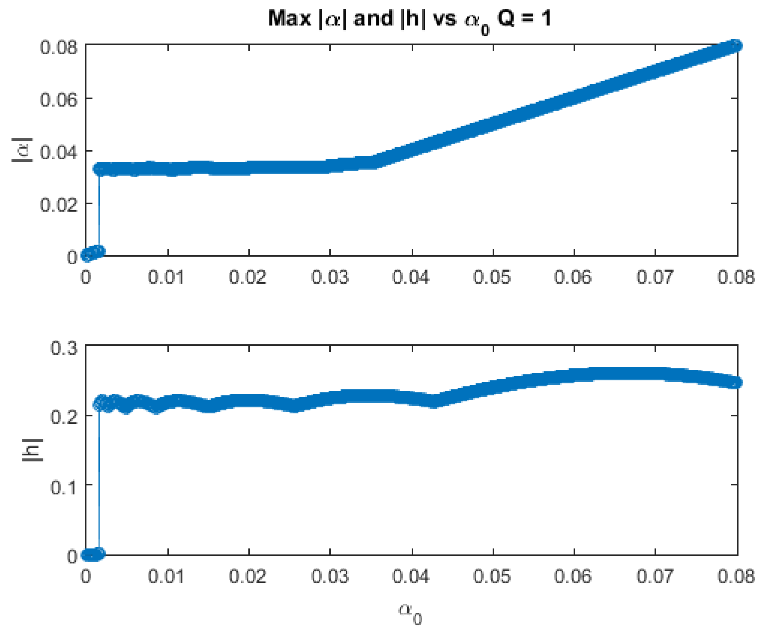
$$\begin{aligned} \alpha_0 &= 0 : 0.001 : 0.08 \\ \text{tspan} &= [0\ 60] \\ \Delta t &= 10^{-3} \end{aligned}$$

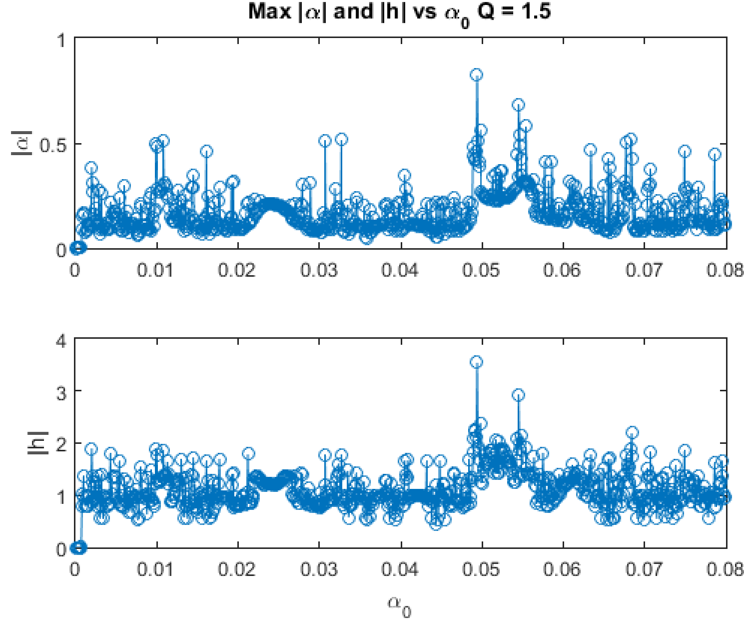
The α array was chosen to ensure that all possible oscillations were found. For each α_0 we compute the solution for $Q = 1$ and $Q = 1.5$ then find the maximum of the absolute value of the height and pitch. The results of this process are plotted below.





If we refine the α array such that $\Delta\alpha$ is now 10^{-4}, we find the following plots.





Thus we can see that the α refinement does not change the results for $Q = 1$ but does expose a few more instabilities for $Q = 1.5$.

The results show that for $Q = 1$ the maximum deflections are fairly predictable. The maximum $|\alpha|$ appears to be equal to around 0.035 for $\alpha_0 < 0.035$ and then linearly increases with slope 1. This indicates that the system starts at α_0 and then settles to smaller values of α for large values of the initial pitch. The maximum value of $|h|$ is approximately constant for all initial pitch values, with a $|h|$ of approximately 0.25. With our initial assumption of a maximum error of 0.01, this means that the magnitude of h can be expected to fall around $0.25 \pm 4\%$. This is an acceptable level of error for most engineering approximations.

For $Q = 1.5$ the maximum $|\alpha|$ and $|h|$ vary widely. Here we see the instabilities in the wing when it is operating near the never-exceed speed. This makes sense because the never-exceed speed is defined as the point where increasing speed will cause structural damage to the aircraft. Thus, we expect the solutions at V_{NE}, $Q = 1.5$, to be large in magnitude. The plots show that the peak pitch is around 0.5 radians and the peak plunge is around $2\bar{c}$. The widely varying nature of the $|\alpha|$ and $|h|$ plots show that there are definitely values of α_0 for which the system is excited more than others.

For these results, we have a high confidence in their accuracy. This is due to the analyses performed for part 1 of this project and the subsequent choices of Δt, tolerance, and scheme. For this analysis, we have chosen the most accurate scheme. For this scheme we have chosen an efficient yet accurate time step size and a suitable tolerance for the Newton iteration. Thus, we are confident that the solutions presented contain minimal numerical error and reflect the true characteristics of the system.

By examining the results from the BDF-2 Δt sensitivity analysis we expect the following errors for $\Delta t = 1 \times 10^{-7}$ and a tolerance of 1×10^{-6}.

	$Q = 1$	$Q = 1.5$
α	1.0×10^{-7}	1.0×10^{-3}
h	1.0×10^{-6}	1.0×10^{-2}

These are definitely tolerable errors, increasing our confidence in our results.

2 Problem 2

We now examine an advection system. The governing equation is given by

$$\frac{\partial u}{\partial t} = \frac{\partial c_x(x,y)u}{\partial x} + \frac{\partial c_y(x,y)u}{\partial y} \tag{7}$$

$$\tag{8}$$

where $u(x, y, t)$ is the density, $c_x(x, y)$ is the flow field in the x direction, and $c_y(x, y)$ is the flow field in the y direction.

We solve this equation over the domain $0 \leq x \leq 1$ and $0 \leq y \leq 1$. The boundary conditions are given by

$$u(x, y, 0) = 0$$
$$u(0, y, t) = u(x, 0, t) = \sin(\pi t)$$

We use a Forward Time, Backward Space (FTBS) method to solve this system. The Backward Space discretization gives

$$\left.\frac{\partial U}{\partial x}\right|_{i,j}^{n} = \frac{c_x U_{i,j}^n - c_x U_{i-1,j}^n}{\Delta x} \tag{9}$$

$$\left.\frac{\partial U}{\partial y}\right|_{i,j}^{n} = \frac{c_y U_{i,j}^n - c_y U_{i,j-1}^n}{\Delta y} \tag{10}$$

The Forward Time discretization is given by

$$\left.\frac{\partial U}{\partial t}\right|_{i,j}^{n} = \frac{U_{i,j}^{n+1} - U_{i,j}^n}{\Delta t} \tag{11}$$

For these equations $\Delta x = \Delta y = 1/512$ and $\Delta t = 1/1024$ as defined in the problem statement.

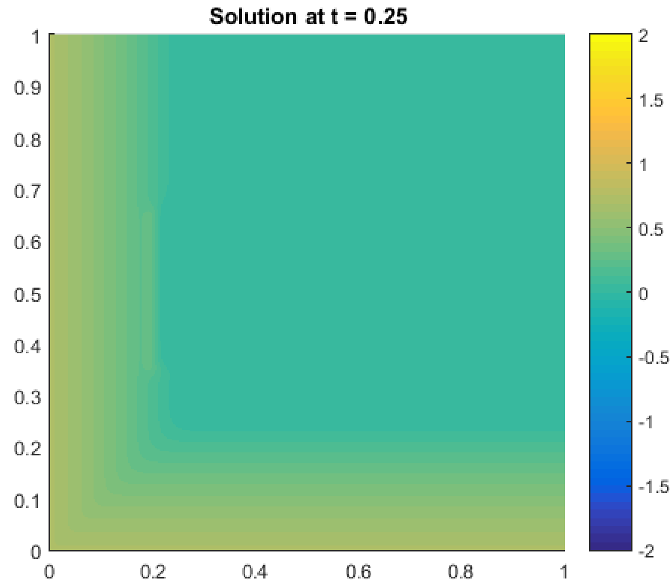
We now set about implementing this scheme. We start with the baseline code that was given with the problem statement. This code loads c_x, c_y, initializes u, and computes the x spacial vector for plotting the results.

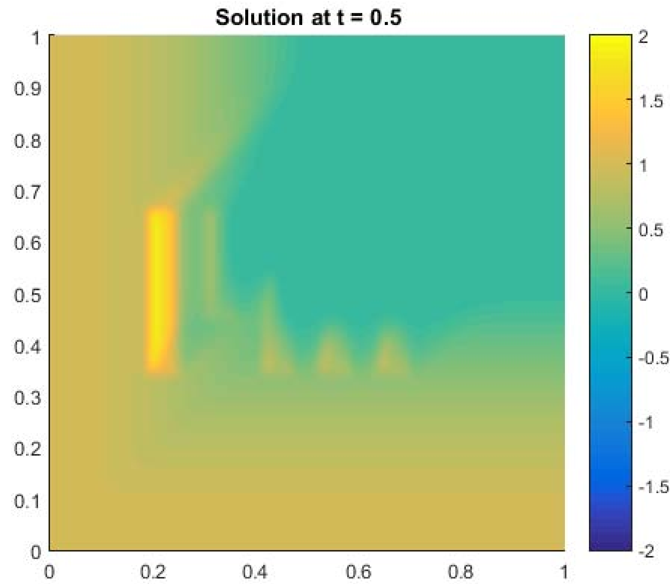
Once the constants have been defined, the derivatives can be computed. We implement equations (9), (10), and (11) in a loop that breaks out at specified values of t. This allows us to check the solution at intermediate points in the timespan. We use the following MATLAB syntax for the computations.

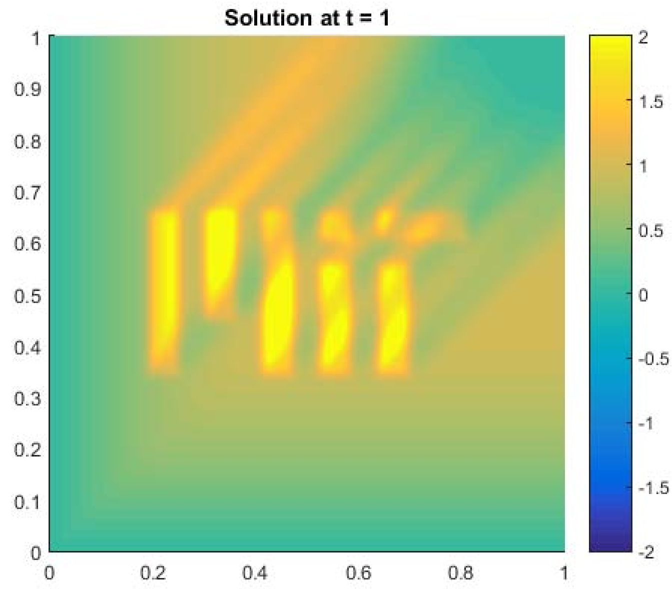
$$\frac{\partial u}{\partial x} = \frac{c_x.*u - \text{circshift}(c_x.*u, [0, 1])}{\Delta x}$$
$$\frac{\partial u}{\partial y} = \frac{c_y.*u - \text{circshift}(c_y.*u, [1, 0])}{\Delta y}$$
$$\frac{\partial u}{\partial t} = -\frac{\partial u}{\partial x} - \frac{\partial u}{\partial y}$$
$$u = u + \Delta t \frac{\partial u}{\partial t}$$

We enforce the boundary conditions by forcing $u(:,1) = u(1,:) = \sin(\pi t)$. We then update the time to the new time $t = t + \Delta t$ and repeat until we reach the final time. In this way we compute the advection using the FTBS scheme.

We write the code such that intermediate solutions are plotted at $t = 0.25$ and $t = 0.5$ in addition to the solution at $t = 1$. The solutions are plotted below.







From these results, we can clearly see that the heat advects toward the upper right corner of our spatial domain. As time progresses the "0" boundary, along the left wall and the bottom, heat up and then re-cool, as is expected due to the sinusoidal nature of the boundary condition. In time, the MIT in the middle becomes visible. This is because the heat leaves this area more slowly, as defined by the c_x and c_y flow field data.

A AeroVibe.m

```
function [ sdot ] = AeroVibe( s )
%AeroVibe Compute derivatives for elastic wing
%   compute derivatives of pitch and height for Aeroelastic wing model
%   Global variable Q must be defined for this code to compile
%   s = [a, a_dot, h, h_dot]
%   sdot = [a_dot, a_dotdot, h_dot, h_dotdot]


global Q

Mhh = 1;
Mha = 0.625;
Maa = 1.25;
Mah = 0.25;
Dh = 0.1;
Da = 0.25;
Kh = 0.2;
Ka = 1.25;
kNL = 10;

a = s(1);
a_dot = s(2);
h = s(3);
h_dot = s(4);

L = Q*a;
M = -0.7*Q*a;

h_dotdot = (Mha/Maa*(Da*a_dot + Ka*(1+kNL*h^2)*a + M)...
    - Dh*h_dot - Kh*h - L)/(Mhh - Mah*Mha/Maa);
a_dotdot = (Mah/Mhh*(Dh*h_dot + Kh*h + L)...
    - Da*a_dot - Ka*(1+kNL*h^2)*a - M)/(Maa - Mah*Mha/Mhh);

sdot = [a_dot; a_dotdot; h_dot; h_dotdot];
```

B AeroVibeJacobian.m

```
function [ jac ] = AeroVibeJacobian( s )
%AEROVIBEJACOBIAN Computes Jacobian for an Aeroelastic wing system
%   compute Jacobian matrix
%
%   Global variable Q must be defined for this code to compile
%   s = [a, a_dot, h, h_dot] --- Current state
%
%   Jacobian = [ adot_a adot_adot adot_h adot_hdot;
%                adotdot_a adotdot_adot adotdot_h adotdot_hdot;
%                hdot_a hdot_adot hdot_h hdot_hdot;
%                hdotdot_a hdotdot_adot hdotdot_h hdotdot_hdot];

global Q

Mhh = 1;
Mha = 0.625;
Maa = 1.25;
Mah = 0.25;
Dh = 0.1;
Da = 0.25;
Kh = 0.2;
Ka = 1.25;
kNL = 10;

a = s(1);
a_dot = s(2);
h = s(3);
h_dot = s(4);

adot_a = 0;
adot_adot = 1;
adot_h = 0;
adot_hdot = 0;

adotdot_a = (Mah/Mhh*Q - Ka*(1+kNL*h^2) + 0.7*Q)/(Maa - Mah*Mha/Mhh);
adotdot_adot = -Da/(Maa - Mah*Mha/Mhh);
adotdot_h = (Mah/Mhh*Kh - Ka*kNL*2*h*a)/(Maa - Mah*Mha/Mhh);
adotdot_hdot = (Mah/Mhh*Dh)/(Maa - Mah*Mha/Mhh);

hdot_a = 0;
hdot_adot = 0;
hdot_h = 0;
hdot_hdot = 1;
```

```
hdotdot_a = (Mha/Maa*(Ka*(1+kNL*h^2) - 0.7*Q) - Q)/(Mhh - Mah*Mha/Maa);
hdotdot_adot = (Mha/Maa*Da)/(Mhh - Mah*Mha/Maa);
hdotdot_h = (Mha/Maa*(Ka*kNL*2*h*a) - Kh)/(Mhh - Mah*Mha/Maa);
hdotdot_hdot = -Dh/(Mhh - Mah*Mha/Maa);

jac = [ adot_a adot_adot adot_h adot_hdot;
        adotdot_a adotdot_adot adotdot_h adotdot_hdot;
        hdot_a hdot_adot hdot_h hdot_hdot;
        hdotdot_a hdotdot_adot hdotdot_h hdotdot_hdot];
end
```

C ForwardEuler.m

```
function [ t, x ] = ForwardEuler( funcName, x_0, tspan, dt )
%FORWARDEULER Compute a Forward Euler scheme
%   [ t, x ] = ForwardEuler( funcName, x_0, tspan, dt )
%   Takes in initial state and runs a Forward Euler scheme over tspan with
%   dt size time steps.  Outputs the solution [x] for all time [t]
%
%   FORWARD EULER: v_n+1 = v_n + dt*f(v_n)
%--------------------------------------------------------------------------
%   INPUTS
%       funcName [string] - filename of function for computing the
%           derivative of the state
%       x_0 [vector] - input initial state
%       tspan [T0 TFINAL] - define initial and final time
%       dt - time step size

% define time span for calculation
t = tspan(1):dt:tspan(2);

% create function handle from filename
fh = str2func(funcName);

% initialize the output vector
x = zeros([length(x_0) length(t)]);

% input initial conditions
x(:,1) = x_0;

% run a Forward Euler scheme
for i = 1:length(t)-1
    xdot = fh(x(:,i));
    x(:,i+1) = x(:,i) + dt*xdot;
end

end
```

D Midpoint.m

```
function [ t, x ] = Midpoint( funcName, x_0, tspan, dt )
%MIDPOINT Compute a Midpoint scheme
%   [ t, x ] = Midpoint( funcName, x_0, tspan, dt ) computes x(i+1) using a
%   Midpoint scheme.  Outputs the solution [x] for all time [t]
%
%   MIDPOINT: v_n+1 = v_n-1 + 2*dt*f(v_n)
%-----------------------------------------------------------------------
%   INPUTS
%       funcName [string] - filename of function for computing the
%           derivative of the state
%       x_0 [vector] - input initial state
%       tspan [T0 TFINAL] - define initial and final time
%       dt - time step size

% compute time span
t = tspan(1):dt:tspan(2);

% create funtion handle for computing the state derivative
fh = str2func(funcName);

% initialize the solution matrix
x = zeros([length(x_0) length(t)]);

% set the initial conditions
x(:,1) = x_0;

% compute the second time step using Forward Euler
xdot = fh(x(:,1));
x(:,2) = x(:,1) + dt*xdot;

% run the Midpoint scheme
for i = 2:length(t)-1
    xdot = fh(x(:,i));
    x(:,i+1) = x(:,i-1) + 2*dt*xdot;
end

end
```

E BDF2.m

```
function [ t, x ] = BDF2( funcName, funcJac, x_0, tspan, dt, tol )
%BDF2  Solve stiff differential equations, 2nd Order, Backward step
%   [ t, x ] = BDF2( funcName, funcJac, x_0, tspan, dt, tol )
%   Solve an implicit system using a second order, backward differentiation
%   scheme.  Returns the solution matrix [x] for all time t in the time
%   vector [t]
%
%   BDF-2: v_n+1 = 4/3*v_n - 1/3*v_n-1 + 2/3*dt*f(v_n+1)
%-------------------------------------------------------------------------
%   INPUTS
%       funcName [string] - function that computes derivative of state
%       funcJac [string] - function that computes Jacobian of state
%       x_0 [vector] -
%       tspan [T0 TFINAL] - define initial and final time
%       dt - define time step size
%       tol - define tolerance size. Used for calculating residual of
%             system

% define time span
t = tspan(1):dt:tspan(2);

% create function handles for derivative and Jacobian functions
f = str2func(funcName);
fJac = str2func(funcJac);

% initialize solution matrix
x = zeros([length(x_0) length(t)]);

% input initial conditions
x(:,1) = x_0;

% compute the second time step in order to run the 2-step scheme
xdot = f(x(:,1));
x(:,2) = x(:,1) + dt*xdot;

% initialize the Identity matrix to the size of the system
I = eye(length(x(:,1)));

% run the BDF-2 scheme to compute x(:, i+1)
for i = 2:length(t)-1

    % set initial guess to previous time step
    w = x(:,i);
```

```
    % compute initial residual
    R = w - 4/3*x(:,i) + 1/3*x(:,i-1) - 2/3*dt*f(w);

    % solve the implicit system until the desired tolerance is met
    while max(abs(R)) > tol

        % compute Jacobian
        df = fJac(w);

        % compute derivative of residual
        dR = I - 2/3*dt*df;

        % solve for dw
        dw = -dR\R;

        % update the guess of x(:, i+1)
        w = w + dw;

        % compute the new residual
        R = w - 4/3*x(:,i) + 1/3*x(:,i-1) - 2/3*dt*f(w);
    end

    % save x(:,i+1)
    x(:,i+1) = w;
end

end
```

F Part 2

```
% Alex Feldstein

% 16.90 Project 1 part 2: Advection
% Model an advection system with a given flow field which spells 'MIT'

clear all;
close all;

% load Cx and Cy from disk
Cx = load('Cx.txt');
Cy = load('Cy.txt');

% set the initial condition
u = zeros(size(Cx));

% set dt as given in the problem statement
dt = 1/1024;

% compute dx and dy
dx = 1/(length(Cx(:,1))-1);
dy = dx;

% compute the x and y spacial vectors
x = dx * [0:(size(Cx,1)-1)];

% initialize t
t = 0;

% compute the solution for the first time interval 0 <= t <= 0.25
while t <= 0.25
    % implemtent a Bacward Space scheme
        dudx = (Cx.*u - circshift(Cx.*u, [0,1]))/dx;
    dudy = (Cy.*u - circshift(Cy.*u, [1,0]))/dy;

    % use the advection equation
    dudt = -dudx + -dudy;

    % implement a forward time scheme
    u = u + dt*dudt;

    % enforce the boundary condition
    u(1,:) = sin(t * pi);
    u(:,1) = sin(t * pi);
```

```
    % compute the new time
    t = t + dt;
end

% plot the solution at t = 0.25
figure;
surf(x, x, u);
shading interp;
caxis([-2,2]);
view(2);
axis equal;
axis([0,1,0,1])
title('Solution at t = 0.25')
colorbar;
drawnow;

% run the solution over the second time interval 0.25 < t <= 0.5
while t <= 0.5
    dudx = (Cx.*u - circshift(Cx.*u, [0,1]))/dx;
    dudy = (Cy.*u - circshift(Cy.*u, [1,0]))/dy;
    dudt = -dudx + -dudy;
    u = u + dt*dudt;

    % enforce the boundary condition
    u(1,:) = sin(t * pi);
    u(:,1) = sin(t * pi);

    t = t + dt;

end

% plot the solution at t = 0.5
figure;
surf(x, x, u);
shading interp;
caxis([-2,2]);
view(2);
axis equal;
axis([0,1,0,1])
title('Solution at t = 0.5')
colorbar;
drawnow;

% run the solution over the third time interval 0.5 < t <= 1
```

```
while t <= 1
    dudx = (Cx.*u - circshift(Cx.*u, [0,1]))/dx;
    dudy = (Cy.*u - circshift(Cy.*u, [1,0]))/dy;
    dudt = -dudx + -dudy;
    u = u + dt*dudt;

    % enforce the boundary condition
    u(1,:) = sin(t * pi);
    u(:,1) = sin(t * pi);

    t = t + dt;

end

% plot the solution at t = 1
figure;
surf(x, x, u);
shading interp;
caxis([-2,2]);
view(2);
axis equal;
axis([0,1,0,1])
title('Solution at t = 1')
colorbar;
drawnow;
```

MIT OpenCourseWare
http://ocw.mit.edu

16.90 Computational Methods in Aerospace Engineering
Spring 2014

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.