Aeroelastic Analysis of Hypersonic Double-wedge Lifting Surface

Koorosh Gobal[1], Anusha Anisetti[1], and Vana Naga Samyuktha Nuthy[1]

[1]Department of Mechanical and Materials Engineering, Wright State University

Abstract

1 Introduction

Nonlinear oscillations problem are important phenomena in physical science, mechanical structural analysis and many other engineering application. Nonlinear systems display different behaviours than linear systems. They can exhibit

1. multiple steady state solutions, stable and unstable, depending on the control parameters (bifurcation),
2. response at frequencies other than forcing frequency
3. irregular motions that are extremely sensitive to the initial conditions (chaos).

Therefore, studying nonlinearieties is of extreme importance. As most of the real systems exhibit nonlinear responses due to material and geometric nonlinearities, predicting their response can become a critical task. There are different approximation techniques to determine the steady state solution of these systems in time domain. Drawback of such time-domain methods is the excessive need for computational time and resources due to expensive time integrations involved. This can become a major bottle neck in design space exploration efforts of such systems where mutiple solutions for different configurations are needed. Therefore, there is a need for techniques that can predict the limit cycle oscillations of nonlinear systems without the need to expensive time integration methods. Harmonic Balance method is one of these approaches that uses frequency domain analysis to calculate the steady-state response of the system.

1.1 Harmonic Balance Method

Harmonic Balance Method assumes a truncated Fourier series as the solution to a nonlinear system.

$$x(t) = a_0 + \sum_{n=1}^{N} (a_n \cos(n\omega_0 t) + b_n \sin(n\omega_0 t)) \tag{1}$$

where ω_0 is the angular frequency with time period T of the solution, N is the number of harmonics, a_n and b_n are the coefficients of Fourier transform, and t is time. The assumed solution is substituted into the equation of motion to determine the Fourier coefficients to form an approximate closed form solution. There is no rule of thumb for choosing N, as a matter of fact this is a trial and error process. If it is known that the system is going to exhibit high frequency oscillations, the number of harmonics (N) need to be increases to be able to capture the response.

1.1.1 Advantages

- For some equations, a first-order truncated Fourier series can provide accurate results, especially as measure in terms of percentage error for the angular frequency ω
- Provides a close to accurate solution to steady-state responses.
- Simulation cost is much less compared to the equivalent time integration for systems with few (less than 100) degree of freedom.

1.1.2 Disdvantages

- when applied to a system white many degrees of freedom, the HB needs to be done for each of the DOF. This can become extremely expensive.
- The resulting matrices for HB are not sparse anymore. Storing these can become problematic.

In this report, Harmonic balance methodology is first build on closed form equation for structural response of linear equations and then extended to basic non-linear equations and non-linear equations with parametric excitations. The results are compared with time integrated solutions of the corresponding systems.

2 Numerical Approach

To solve a general ordinary nonlinear differential using the Harmonic Balance (HB) method, we first define the governing equations in terms of a residual function as follows

$$\mathcal{R} = F(\ddot{x}, \dot{x}, x, f) \tag{2}$$

Where $\ddot{x}$ is the acceleration, $\dot{x}$ is velocity, x is displacement, and f are the rest of variable in the equation of motion. F is the general functional form of the governing equations. We then assume a solution $\bar{x}$. If $\bar{x}$ is indeed the solution of the equation of motion, when substituting it in the residual equation of (15) we should get zero. However, since it is an assumed solution, the residual won't be zero. By updating $\bar{x}$ is an optimization loop to minimize $\mathcal{R}^2$, we can get the solution of Equation (15). In this process, it is required to calculate $\ddot{x}$ and $\dot{x}$ for $\mathcal{R}$. This is done in frequency domain.

The variable x in the time domain can be approximated as follows in the frequency domain.

$$x(t) = \sum_{k=-\infty}^{\infty} X_k \cdot e^{i\omega_0 k t} \quad , \quad \omega_0 = \frac{2\pi}{T} \tag{3}$$

The time derivatives are calculated by differentiating above equation with respect to time.

$$\dot{x}(t) = \sum_{k=-\infty}^{\infty} i\omega_k X_k \cdot e^{i\omega_k t} \tag{4a}$$

$$\ddot{x}(t) = \sum_{k=-\infty}^{\infty} -\omega_k^2 X_k \cdot e^{i\omega_k t} \tag{4b}$$

By substituting Equations (3), (4a), and (4b) into Equation (15), we can minimize the residual based on a guessed displacement. The flow chart for this is shown in Figure 1.

To transfer the time domain data for frequency domain we use `numpy.fft.fft` function and for converting the frequency domain data back to time domain we used `numpy.fft.ifft`.

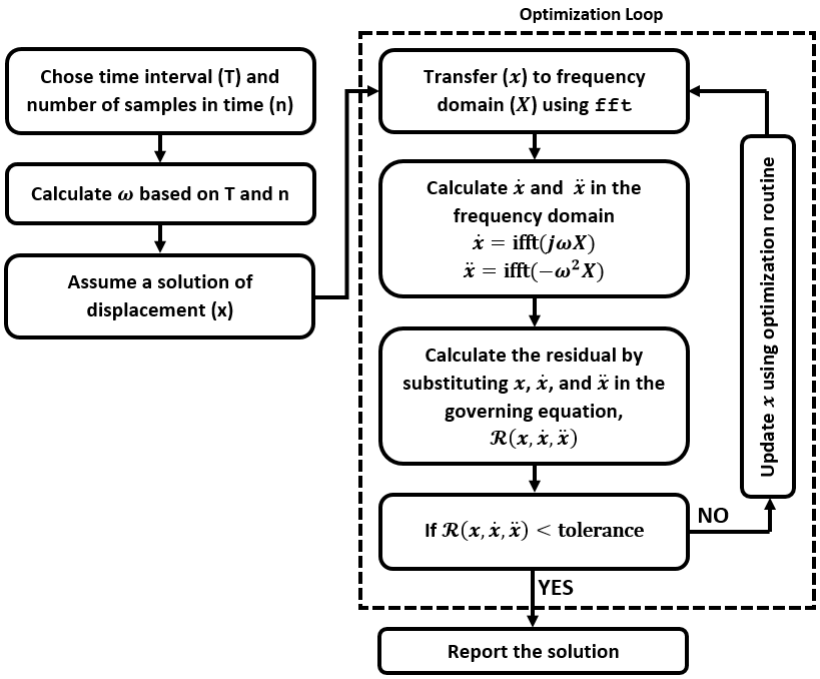


Figure 1: Flowchart for numerical approach.

We sampled the frequency domain using 19 points. The initial guess for displacement is selected as a vector of ones, `numpy.ones`. It should be noted that this is a vector of displacement in time. The minimization of residual is done using python `scipy.optimize.minimize` function using `SLSQP` method. This process is shown in Figure 1.

This process can be summarized in the following bullet points

- An initial guess for response in time domain.
- Transform the initial guess into frequency domain.
- Obtain the derivatives of the responses in the frequency domain.
- Transform the derivatives back into time domain
- Minimizing the cost function of the residuals from the equations of motion
- Re-iterate until minimization algorithm is below it's convergence criteria tolerance.

To verify the result of HB method, we used numerical integration to calculate the solution of Equation (15). It should be noted that HB is capable of capturing the steady-state response of the system. Therefore, it is required to let the transient response of the time integration to die of before comparing the results. These are shown in next section.

3 Demonstration Results

In this section, we apply the method of harmonic balance to different linear and non-linear problems. For the cases where analytical results are not available, we use numerical time integration to verify the results where it shows a good comparison. It should be noted that since the harmonic balance method is used to calculate the limit cycle oscillations of the system. Therefore, it is needed to let the transient part of time integration to die out. Only after this point, the two results match.

3.1 Linear MCK System

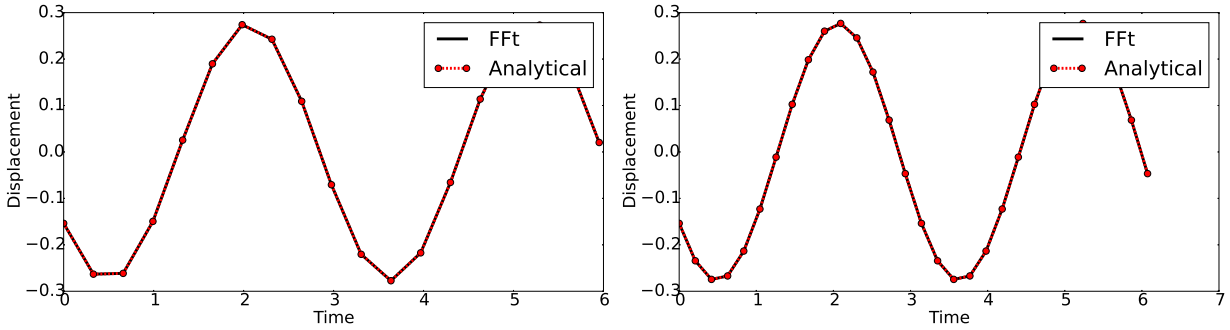
For the first demonstration case we looked at the forced linear damped oscillation of a single degree of freedom system. The governing equation is defined in Equation (5)

$$\ddot{x} + \dot{x} + x = \sin 2t \tag{5}$$

This equation has a closed form solution that we can use to verify the harmonic balance result. The solution for Equation (5) is calculated as

$$x(t) = -0.23 \sin 2t - 0.15 \cos 2t \tag{6}$$

The comparison between the HB and analytical result for Equation (5) is shown in Figure 2 for different number of harmonics in HB method. As can be seen here, the HB results matches perfectly with analytical solution of the system.



(a) n = 19 (b) n = 30

Figure 2: Comparison between HB and analytical result.

The convergence results for the residual is shown in Figure 3. As can be seen here, the value of residual reaches zero at the end of the optimization process. The flat lines represents the finite difference steps required to calculated the Jacobian. This Jacobian is used to define the direction the optimizer goes to update the design variables, vector x in time. As can be seen here, if the simulation takes a long time to converge, the finite difference step is going to take a significant amount of time and resources.

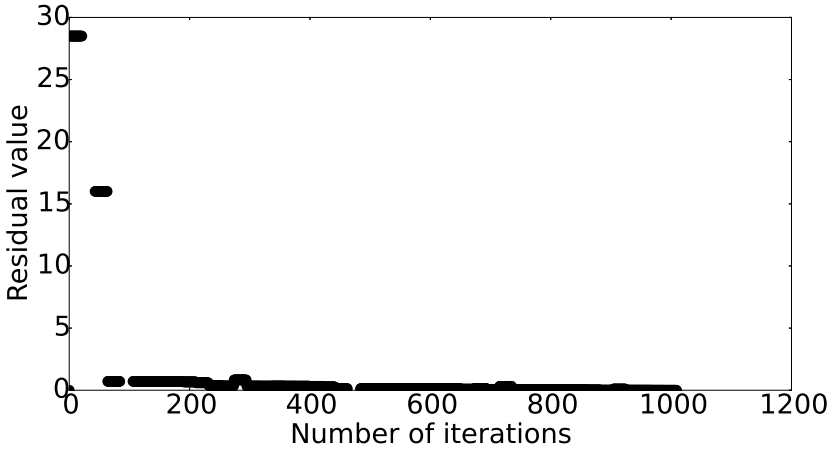


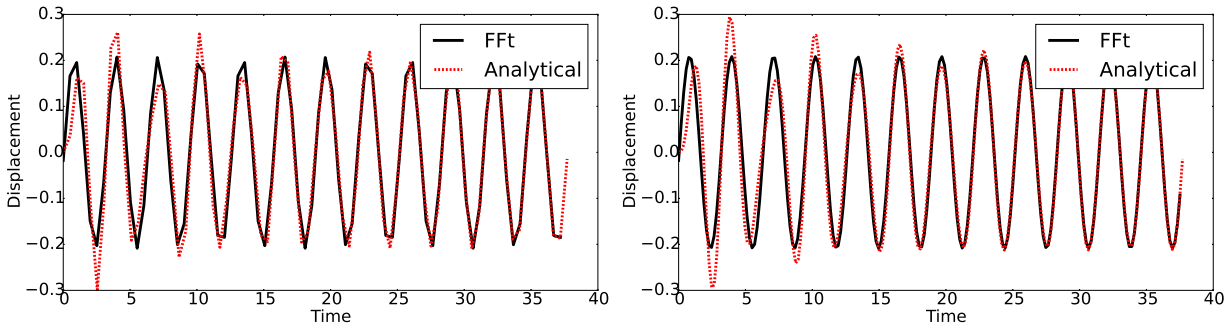
Figure 3: Convergence plot.

3.2 Nonlinear Oscillator

For the second demonstration example, we looked at problem 2.45 on page 140 of Applied Nonlinear Dynamics (Nayfeh) book. The governing equation for this problem is shown in Equation (7)

$$\ddot{x} + 2\mu\dot{x} + \frac{g}{R}\sin x - \alpha^2 \sin x \cos x = \sin 2t \tag{7}$$

In above equation, we chose $\mu = 0.1$, $g = 9.81$, $R = 1.0$, and $\alpha = 1$. We used `odeint` function in `python` for time integration to verify the results of HB method. This is shown in Figure 4. As shown here, HB is in good agreement with the numerical solution at both low and high order sampling. The response converges and coincides with analytical solution as the transient part of the solution dies out.



(a) n = 75 (b) n = 199

Figure 4: Comparison between HB and analytical result.

The convergence results for the residual is shown in Figure 5. As can be seen here, the value of residual reaches zero at the end of the optimization process. The flat lines represents the finite difference steps required to calculated the Jacobian.

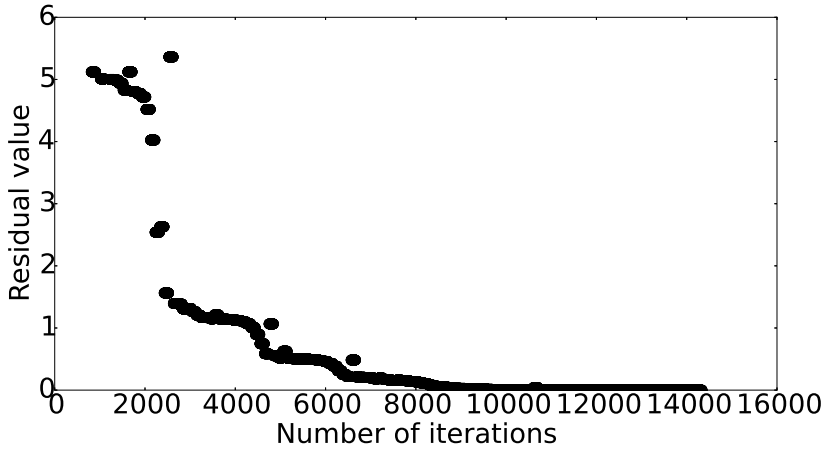


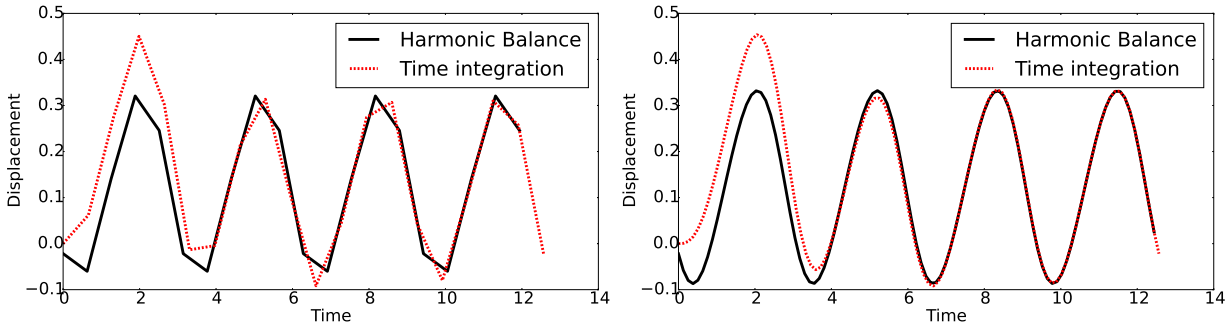
Figure 5: Convergence plot.

3.3 Parametrically excited Duffing Oscillator

We looked at the applicability of HB in handling the nonlinear, force, parametrically excited Duffing oscillator. The governing equation of this system is shown in Equation (8).

$$\ddot{x} + x + \epsilon\left[2\mu\dot{x} + \alpha x^3 + 2kx\cos\omega t\right] = \sin 2t \tag{8}$$

We chose $\epsilon = 1.0$, $\mu = 1.0$, $\alpha = 1.0$, $k = 1.0$, and $\omega = 2.0$ for the system parameters. Like the previous example, we used time integration to verify the results of HB method as shown in Figure 6. As seen here, similar convergence trend is observed for parametrically excited nonlinear oscillating system for different number of harmonics.



(a) n = 20 (b) n = 99

Figure 6: Comparison between HB and analytical result.

The convergence results for the residual is shown in Figure 7. As can be seen here, the value of residual reaches zero at the end of the optimization process. The flat lines represents the finite difference steps required to calculated the Jacobian.

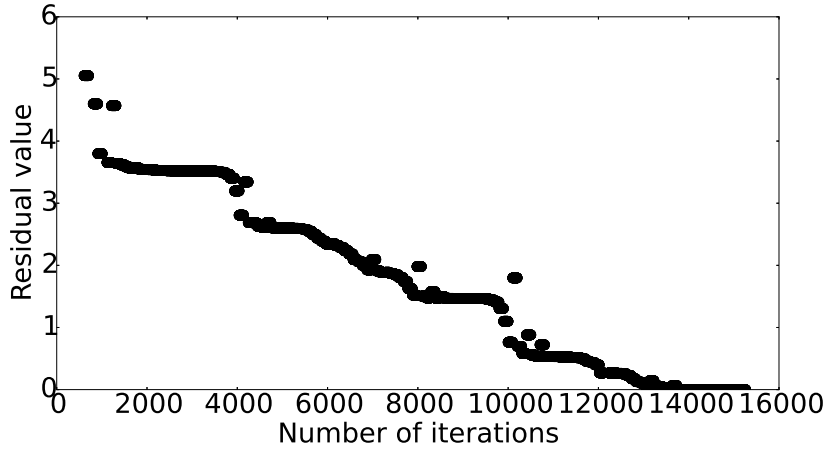


Figure 7: Convergence plot.

3.4 Double Pendulum

Adapting Harmonic balance method for a two-degree of freedom system, a double pendulum, with θ_1 and θ_2 DOF is considered. If m_1, m_2, l_1 and l_2, c_1 and c_2 are masses, lengths and damping coefficients of pendulum 1 and pendulum 2 respectively, then the equations of motion are

$$(m_1+m_2)l_1\ddot{\theta_1}+m_2l_2\ddot{\theta_2}\cos(\theta_1-\theta_2)+m_2l_2\dot{\theta_2}^2\sin(\theta_1-\theta_2)+(c_1+c_2)(l_1^2)\dot{\theta_1}+c_2l_1l_2\dot{\theta_2}+g(m_1+m_2)\sin(\theta_1)=0 \tag{9}$$

$$m_2l_2\ddot{\theta_2}+m_2l_1\ddot{\theta_1}\cos(\theta_1-\theta_2)-m_2l_1\dot{\theta_1}^2\sin(\theta_1-\theta_2)+c_2\dot{\theta_2}+c_2l_1l_2\dot{\theta_1}+c_2(l_2^2)\dot{\theta_2}+m_2g\sin(\theta_2)=\sin(\omega t) \tag{10}$$

The equations of motion are written in state space form

$$\dot{u1}=u2 \tag{11}$$

$$\dot{u2}=\frac{ed-bf}{ad-bc} \tag{12}$$

$$\dot{v1}=v2 \tag{13}$$

$$\dot{v2} = \frac{af - ce}{ad - bc} \tag{14}$$

where $a = (m_1 + m_2)l_1$, $b = m_2 * l_2 * cos(u_1 - v_1)$, $c = m_2 * l_1 * cos(u_1 - v_1)$, $d = m_2 l_2$, $e = -m_2 l_2 v_2^2 sin(u_1 - u_2) - g(m_1 + m_2)sin(u_1) + (c_1 + c_2)(l_1^2)u_2 + c_2 l_1 l_2 v_2$ and $f = m_2 l_1 u_2^2 sin(u_1 - v_1) - m_2 g sin(v_1) + sin(\omega t) + c_2 v2 + c_2 l_1 l_2 u_2 + c_2(l_2^2)v_2$

The results show chaotic behaviour of the double pendulum as expected.

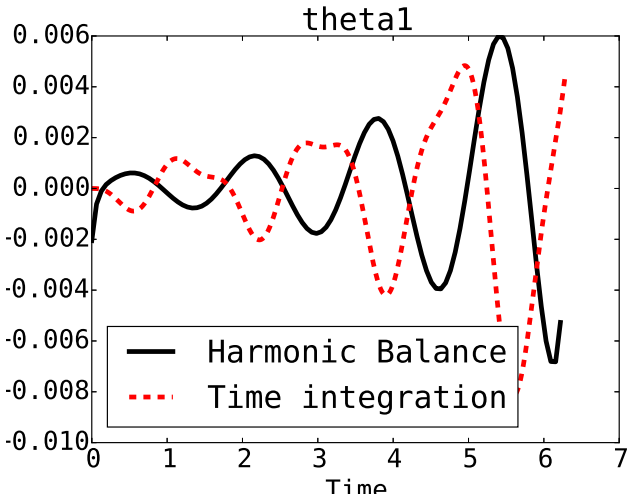


Figure 8: Response θ_1

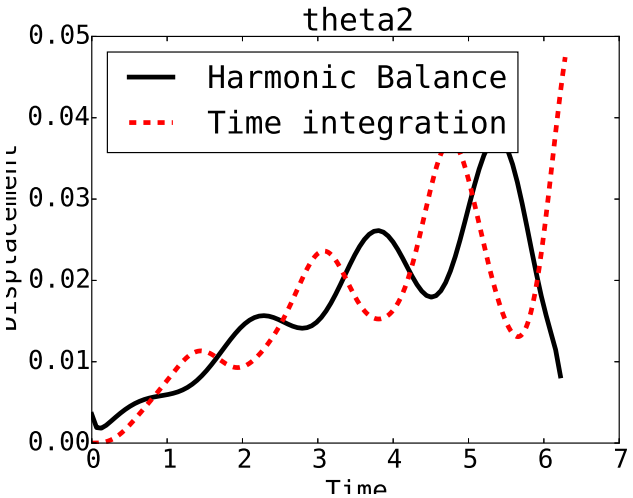


Figure 9: Response θ_2

3.5 Hypersonic Flutter

In this work, piston theory [1] is applied to a double-wedged lifting surface with pitching degree of freedom as shown in Figure 10. This work uses third-order piston theory to compute unsteady effects behind shock waves and expansion fans and a stiffening spring to the structure. This method of computing unsteady aerodynamic effects delivered highly accurate results when compared to computational fluid dynamics solutions [2].

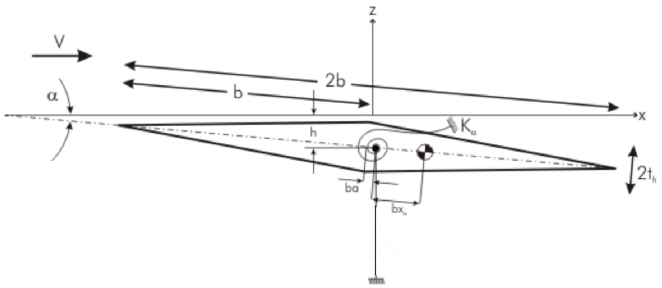


Figure 10: Double-wedge lifting surface.

The governing equation can be written as

$$m\ddot{\theta} + k\theta = f(\theta, t) \tag{15}$$

where m is the mass of airfoil, k is the stiffness, θ is the angle of attack and f is the aerodynamic load. We use the Piston Theory to calculate the aerodynamic load acting on the airfoil. We used the piston theory to calculate the pressure distribution on the airfoil.

$$\frac{P(x,t)}{P_\infty} = \left(1 + \frac{\gamma - 1}{2}\frac{v_n}{a_\infty}\right)^{\frac{2\gamma}{\gamma-1}} \tag{16}$$

where $P(x,t)$ is the pressure at point x on the airfoil, P_∞ is the free stream pressure, γ is the ratio of specific heats and a_∞ is the speed of sound. The third order expansion of Equation (16) results in

$$P(x,t) = P_\infty \left[1 + \gamma\frac{v_n}{a_\infty} + \frac{\gamma(\gamma+1)}{4}\left(\frac{v_n}{a_\infty}\right)^2 + \frac{\gamma(\gamma+1)}{12}\left(\frac{v_n}{a_\infty}\right)^3\right] \tag{17}$$

v_n is calculated using the following equation.

$$v_n = \frac{\partial Z(x,t)}{\partial t} + V\frac{\partial Z(x,t)}{\partial x} \tag{18}$$

where V is the free stream velocity and $Z(x,t)$ is the position of airfoil surface. The position of airfoil surface can be related to the angle of attack using the following equation:

$$Z(x,t) = M_r\left(\theta(t)\right) Z_0(x) \tag{19}$$

where M_r is the rotation matrix, and $Z_0(x)$ is the initial shape of the airfoil. As can be seen here, the location of surface at time t only depends on the angle of attach at that time, $\theta(t)$. The rotation matrix is defined as

$$M_r = \begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}$$

By expanding Equation (16), and substituting for $Z(x,t)$ from Equation (19), Equation (15) can be rewritten as

$$m\ddot{\theta} + k\theta = \oint_{airfoil} P_\infty \left[\gamma\frac{v_n}{a_\infty} + \frac{\gamma(\gamma+1)}{4}\left(\frac{v_n}{a_\infty}\right)^2 + 1\right] ds \tag{20a}$$

$$v_n = \frac{\partial M_r(\theta)}{\partial t} Z_0(x) + V M_r(\theta)\frac{\partial Z_0(x)}{\partial x} \tag{20b}$$

Equation (20) needs to be solved for θ using Harmonic Balance method. For this problem the operating conditions of the vehicle are selected as follows:

γ	1.4
a_∞	343 m/s
P_∞	10 Pa
V	600 m/s

Table 1: Operating conditions.

For verification, the harmonic balance results are compared with the numerical integration is Figure (11). As shown here, the harmonic balance predicts a static solution ($\theta(t_\infty) = -1.706$) however, the numerical integration predicts oscillation. The interesting point in here is that the time average value of numerical integration is the same as harmonic balance results. This means that the harmonic balance did not have enough frequency content to capture the oscillations. It only captured the constant term.

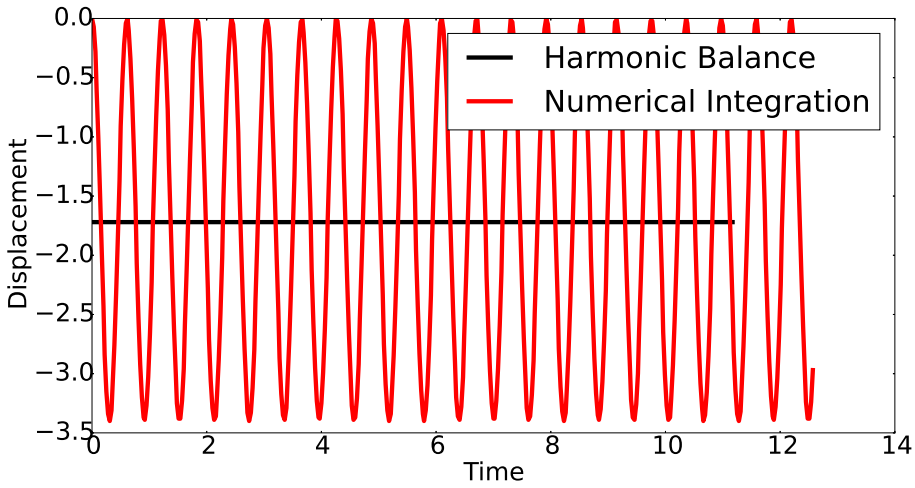


Figure 11: Comparison between HB and analytical result

4 Conclusions

In this project, Harmonic balance method is applied to linear and non-linear equations. The results are compared with the time-integrated responses of these systems. Initially, linear system is solved to validate the results from Harmonic Balance method and time integrated methods and the study is extended to non-linear systems like two-dof system, parametrically oscillated system and systems with flutter responses.

Responses from linear and moderately non-linear systems using Harmonic Balance method converge to the time-integrated response in due course of time. This is because Harmonic balance method results in steady-state reponse and it is required to let the transient response of the time integration to die of before comparing the results. Taking more frequency components into account, Harmonic Balance Method provides better approximation to the response.

For some nonlinear systems that are highly non-linear and behave chaotically such as double pendulum, Harmonic Balance Method solution requires higher harmonics at the cost of computational time. Overall, Harmonic balance method is efficient in obtaining closed form steady-state solution to the non-linear equations.

5 Appendices

```
# Author: Koorosh Gobal
# Python code for 3.1
# ----------------------------------
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.integrate import odeint
# ----------------------------------
N = 19
T = 2*np.pi
t = np.linspace(0, T, N+1)
t = t[0:-1]
f = np.sin(2*t)
Omega = np.fft.fftfreq(N, T/(2*np.pi*N))
x0 = np.ones(N)

xAnalytical = -0.23077 * np.sin(2*t) -0.15385 * np.cos(2*t)
def residual(x):
    X = np.fft.fft(x)
    ddx = np.fft.ifft(np.multiply(-Omega**2, X))
    dx = np.fft.ifft(np.multiply(1j * Omega, X))
    R = ddx + dx + x - f
    R = np.sum(np.abs(np.real(R)))
    R = np.sum(np.abs((R**2)))
    return R
#
print(residual(xAnalytical))
res = minimize(residual, x0, method='SLSQP')
print(residual(res.x))
plt.figure()
plt.plot(t, res.x, 'k',
         t, xAnalytical, 'r--o')
plt.legend(['FFt', 'Analytical'])
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.show()
print(res.jac)
```

```
# Author: Koorosh Gobal
# Python code for 3.2
# ----------------------------------
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.integrate import odeint
# ----------------------------------
# Define system properties
mu = 0.1
g = 9.81
R = 1.0
alpha = 1

N = 99
T = 6*2*np.pi
t = np.linspace(0, T, N+1)
t = t[0:-1]
Omega = np.fft.fftfreq(N, T/(2*np.pi*N))
x0 = np.zeros(N)

# Harmonic Balance method
def residual(x):
    X = np.fft.fft(x)
    ddx = np.fft.ifft(np.multiply(-Omega**2, X))
    dx = np.fft.ifft(np.multiply(1j * Omega, X))
    Residual = ddx + 2 * mu * dx + g / R * np.sin(x)
               - alpha**2 * np.sin(x) * np.cos(x) - f
    Residual = np.sum(np.abs((Residual**2)))
    return Residual
#
res = minimize(residual, x0, method='SLSQP')
xSol = res.x

# Numerical solution
def RHS(X, t=0.0):
    x1, x2 = X
    x1dot = x2
    x2dot = -2 * mu * x2 - g / R * np.sin(x1)
            + alpha**2 * np.sin(x1) * np.cos(x1) + np.sin(2*t)
    return [x1dot, x2dot]
#
ta = np.linspace(0.0, T, N)
sol = odeint(RHS, [0, 0], ta)

plt.figure()
plt.plot(t, res.x, 'k',
         ta, sol[:, 0], 'r--')
plt.legend(['FFt', 'Analytical'])
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.legend(['Harmonic Balance', 'Time integration'], loc='best')
plt.show()
```

```
# Author: Koorosh Gobal
# Python code for 3.3
# ----------------------------------
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.integrate import odeint
# ----------------------------------
epsilon = 1.0
mu = 1.0
alpha = 1.0
k = 1.0
omega = 2.0

N = 99
T = 2*2*np.pi
t = np.linspace(0, T, N+1)
t = t[0:-1]
Omega = np.fft.fftfreq(N, T/(2*np.pi*N))
x0 = np.zeros(N)

# Harmonic Balance method
def residual(x):
    X = np.fft.fft(x)
    dx = np.fft.ifft(np.multiply(1j * Omega, X))
    ddx = np.fft.ifft(np.multiply(-Omega**2, X))
    Residual = ddx + x + epsilon * (2 * mu * dx + alpha * x**3
               + 2 * k * x * np.cos(omega * t)) - np.sin(2 * t)
    Residual = np.sum(np.abs((Residual**2)))
    return Residual
#
res = minimize(residual, x0, method='SLSQP')
xSol = res.x

# Numerical solution
def RHS(X, t=0.0):
    x1, x2 = X
    x1dot = x2
    x2dot = -x1 - epsilon * (2 * mu * x2 + alpha * x1**3
            + 2 * k * x1 * np.cos(omega * t)) + np.sin(2 * t)
    return [x1dot, x2dot]
#
ta = np.linspace(0.0, T, N)
sol = odeint(RHS, [0, 0], ta)
plt.figure()
plt.plot(t, res.x, 'k',
         ta, sol[:, 0], 'r--')
plt.legend(['Harmonic Balance', 'Time integration'], loc='best')
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.show()
```

```
# Author: Koorosh Gobal
# Python code for 3.5
# Main file for flutter analysis
# ----------------------------------
import getShape as gs
import aerodynamic as aero
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.integrate import odeint
# ----------------------------------
gs.wedge(theta = 0, thetaDot = 5)
[Fx, Fy, Mz] = aero.calcLoad()

N = 9
T = 1*2*np.pi
t = np.linspace(0, T, N+1)
t = t[0:-1]
Omega = np.fft.fftfreq(N, T/(2*np.pi*N))
x0 = np.zeros(N)

def residual(x):
    X = np.fft.fft(x)
    dx = np.fft.ifft(np.multiply(1j * Omega, X))
    ddx = np.fft.ifft(np.multiply(-Omega**2, X))
    f = np.zeros(N)
    for ix in range(0, len(x)):
        if np.imag(x[ix]) > 1e-3 or np.imag(dx[ix]) > 1e-3:
            np.disp('You have a problem with imaginary numbers!')
            np.disp([np.imag(x[ix]), np.imag(dx[ix])])
        gs.wedge(theta=np.real(x[ix]), thetaDot=np.real(dx[ix]))
        [Fx, Fy, Mz] = aero.calcLoad()
        f[ix] = Mz
    Residual = ddx + 100 * x - Mz
    Residual = np.sum(np.abs((Residual**2)))
    return Residual

res = minimize(residual, x0, method='SLSQP', options={'maxiter':10000000})
xSol = res.x

# Numerical solution
def RHS(X, t=0.0):
    x1, x2 = X
    gs.wedge(theta=x1, thetaDot=x2)
    [Fx, Fy, Mz] = aero.calcLoad()
    x1dot = x2
    x2dot = -100*x1 + Mz
    return [x1dot, x2dot]

ta = np.linspace(0.0, 1*T, 50*N)
sol = odeint(RHS, [0, 0], ta)

plt.figure()
```

```
plt.plot(t, xSol, 'k',
         ta, sol[:, 0], 'ro')
plt.legend(['Harmonic Balance', 'Numerical Differentiation'])
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.optimize import fmin_slsqp
from scipy.integrate import odeint
# Author: 'Anusha Anisetti
font = {'family' : 'monospace',
        'weight' : 'normal',
        'size'   : 22}
plt.rc('font', **font)
linewidth = 4.0
markersize = 10
# # ------------------------------------------------------------------------
# # 2DOF Mass-Spring-Damper System
# # Define system properties
m1 = 2
m2 = 1
l1 = 1
l2 = 2
g = 32.2
c1 = 1
c2 = .5

N = 99
T = 2*np.pi
t = np.linspace(0, T, N+1)
t = t[0:-1]
p = np.sin(0.2*t)
Omega = np.fft.fftfreq(N, T/(2*np.pi*N))
x0 = np.zeros(N*2)

def residual(x):
    x1 = x[:N]
    x2 = x[N:]

    X1 = np.fft.fft(x1)
    X2 = np.fft.fft(x2)
    dx1 = np.fft.ifft(np.multiply(1j * Omega, X1))
    dx2 = np.fft.ifft(np.multiply(1j * Omega, X2))
    ddx1 = np.fft.ifft(np.multiply(-Omega**2, X1))
    ddx2 = np.fft.ifft(np.multiply(-Omega**2, X2))
    a = (m1+m2)*l1
    b = m2*l2*np.cos(x1-x2)
    c = m2*l1*np.cos(x1-x2)
    d = m2*l2
    e = -m2*l2*dx2**2*np.sin(x1-x2)-g*(m1+m2)*np.sin(x1)+(c1+c2)*(l1**2)*d
    f = m2*l1*dx1**2*np.sin(x1-x2)-m2*g*np.sin(x2)+p+c2*dx2+c2*l1*l2*dx1+c
    R1 = a*ddx1+b*ddx2-e
    R2 = c*ddx2+d*ddx1-f
    Residual = R1**2 + R2**2
    Residual = np.sum(np.abs((Residual)))
    return Residual
```

```
res = minimize(residual, x0)
print(residual(res.x))
xSol = res.x
xSol1 = xSol[:N]
xSol2 = xSol[N:]

# Numerical solution
def RHS(X, t=0.0):
     x11, x12, x21, x22 = X
     x11dot = x12
     x21dot = x22
     a = (m1+m2)*l1
     b = m2*l2*np.cos(x11-x21)
     c = m2*l1*np.cos(x11-x21)
     d = m2*l2
     e = -m2*l2*x22**2*np.sin(x11-x21)-g*(m1+m2)*np.sin(x11)+(c1+c2)*l1**2
     f = m2*l1*x12**2*np.sin(x11-x21)-m2*g*np.sin(x21)+np.sin(0.2*t)+c2*l1
     x12dot = (e*d-b*f)/(a*d-c*b)
     x22dot = (a*f-c*e)/(a*d-c*b)
     return [x11dot, x12dot, x21dot, x22dot]
#
ta = np.linspace(0.0, T, 20*N)
sol = odeint(RHS, [0, 0, 0, 0], ta)
print(sol)

plt.figure()
plt.plot(t, xSol1, 'k',
         ta, sol[:, 0], 'r--',
         lw=linewidth, ms=markersize)
plt.legend(['Harmonic Balance', 'Time integration'], loc='best')
plt.title('theta1')
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.savefig('theta1.eps', format='eps', dpi=1000, bbox_inches='tight')
plt.show()
#
plt.figure()
plt.plot(t, xSol2, 'k',
         ta, sol[:, 2], 'r--',
         lw=linewidth, ms=markersize)
plt.legend(['Harmonic Balance', 'Time integration'], loc='best')
plt.title('theta2')
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.savefig('theta2.eps', format='eps', dpi=1000, bbox_inches='tight')
plt.show()
```

```
# Author: Koorosh Gobal
# Python code for 3.3
# This function calculate airodynamic loads on the wedge
'''
Piston theory is an inviscid unsteady aerodynamic method used
extensively in hypersonic aeroelasticity, which predicts a point function
relationship between the local pressure on a lifting surface and the norma
component of fluid velocity produced by the lifting surface motion.
Here we use the third order expansion of "simple wave" expression for
the pressure on a piston.

This code is based on the following paper:
@article{thuruthimattam2002aeroelasticity,
  title={Aeroelasticity of a generic hypersonic vehicle},
  author={Thuruthimattam, BJ and Friedmann, PP and McNamara, JJ and Powell
  journal={AIAA Paper},
  volume={1209},
  pages={2002},
  year={2002}
}
'''
# -------------------------------------
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
import pdb
# -------------------------------------
def calcLoad():
    gamma = 1.4 # For diatomic gas
    ainfty = 343.0 # Speed of sound (m/s)
    Pinfty = 10.0 # Free stream pressure
    V = 600.0 # Free stream velocity
    Zx = np.loadtxt('coord.txt')[:, 0]
    Zy = np.loadtxt('coord.txt')[:, 1]

    # Read the spatial part of the velocity
    dZdx = np.loadtxt('grad.txt')[:, 1]
    dZdt = np.loadtxt('coordDot.txt')[:, 1]

    # Calculate normal velocity
    vn = dZdt + V * dZdx

    # Calculate pressure
    P = Pinfty + Pinfty * (gamma * vn / ainfty + gamma * (gamma + 1)
                           * (vn / ainfty)**2 / 4 +
                           gamma * (gamma + 1) * (vn / ainfty)**3 / 12)

    # Calculate net force and moment around center of mass
    normalVec = np.loadtxt('normal.txt')
    dA = np.sqrt((Zx[1] - Zx[0])**2 + (Zy[1] - Zy[0])**2)
    Fx = np.multiply(P, normalVec[:, 0]) * dA
    Fy = np.multiply(P, normalVec[:, 1]) * dA
    Rx = -Zx
```

```
Ry = -Zy

Mz = np.zeros(len(Zx))
for ni in range(0, len(Mz)):
    Mz[ni] = np.cross([Rx[ni], Ry[ni]], [Fx[ni], Fy[ni]])

np.savetxt('P.txt', P, '%2.2f')
np.savetxt('Fx.txt', Fx, '%2.2f')
np.savetxt('Fy.txt', Fy, '%2.2f')
Mz = -np.sum(Mz)
Fy = np.sum(Fy)
Fx = np.sum(Fx)

return [Fx, Fy, Mz]
```

```
# Author: Koorosh Gobal
# Python code for 3.3
# This function calculates the shape of wedge at each time step
# and also the velocity at which it is rotating
# ----------------------------------
import matplotlib.pylab as plt
import numpy as np
# ----------------------------------
def wedge(theta = 0, thetaDot = 0):
    vertCoord = np.array([[-5,0],
                          [0,1],
                          [5, 0],
                          [0, -1]])
    N = 3
    n = N + 2
    nodeCoord = np.zeros([4*N, 2])
    nodeGrad = np.zeros([4*N, 2])
    nodeTangent = np.zeros([4*N, 2])
    nodeNormal = np.zeros([4*N, 2])
    # ------------------------------------------------------------------
    nodeCoordX = np.linspace(vertCoord[0, 0], vertCoord[1, 0], n)
    nodeCoordY = (vertCoord[1, 1] - vertCoord[0, 1]) / \
                 (vertCoord[1, 0] - vertCoord[0, 0]) * nodeCoordX
                 + vertCoord[1, 1]
    nodeCoord[:N, 0] = nodeCoordX[1:-1]
    nodeCoord[:N, 1] = nodeCoordY[1:-1]
    nodeGrad[:N, 0] = nodeCoordX[1:-1]
    nodeGrad[:N, 1] = (vertCoord[1, 1] - vertCoord[0, 1]) / \
                      (vertCoord[1, 0] - vertCoord[0, 0]) *
                      np.ones(N)
    nodeTangent[:N, 0] = vertCoord[1, 0] - vertCoord[0, 0]
    nodeTangent[:N, 1] = vertCoord[1, 1] - vertCoord[0, 1]
    # ------------------------------------------------------------------
    nodeCoordX = np.linspace(vertCoord[1, 0], vertCoord[2, 0], n)
    nodeCoordY = (vertCoord[2, 1] - vertCoord[1, 1]) / \
                 (vertCoord[2, 0] - vertCoord[1, 0]) * nodeCoordX
                 + vertCoord[1, 1]
    nodeCoord[N:2*N, 0] = nodeCoordX[1:-1]
    nodeCoord[N:2*N, 1] = nodeCoordY[1:-1]
    nodeGrad[N:2*N, 0] = nodeCoordX[1:-1]
    nodeGrad[N:2*N, 1] = (vertCoord[2, 1] - vertCoord[1, 1]) / \
                         (vertCoord[2, 0] - vertCoord[1, 0]) *
                         np.ones(N)
    nodeTangent[N:2*N, 0] = vertCoord[2, 0] - vertCoord[1, 0]
    nodeTangent[N:2*N, 1] = vertCoord[2, 1] - vertCoord[1, 1]
    # ------------------------------------------------------------------
    nodeCoordX = np.linspace(vertCoord[2, 0], vertCoord[3, 0], n)
    nodeCoordY = (vertCoord[3, 1] - vertCoord[2, 1]) / \
                 (vertCoord[3, 0] - vertCoord[2, 0]) * nodeCoordX
                 + vertCoord[3, 1]
    nodeCoord[2*N:3*N, 0] = nodeCoordX[1:-1]
    nodeCoord[2*N:3*N, 1] = nodeCoordY[1:-1]
    nodeGrad[2*N:3*N, 0] = nodeCoordX[1:-1]
```

```
nodeGrad[2*N:3*N, 1] = (vertCoord[3, 1] - vertCoord[2, 1]) / \
                       (vertCoord[3, 0] - vertCoord[2, 0]) *
                       np.ones(N)
nodeTangent[2*N:3*N, 0] = vertCoord[3, 0] - vertCoord[2, 0]
nodeTangent[2*N:3*N, 1] = vertCoord[3, 1] - vertCoord[2, 1]
# ----------------------------------------------------------------------
nodeCoordX = np.linspace(vertCoord[3, 0], vertCoord[0, 0], n)
nodeCoordY = (vertCoord[0, 1] - vertCoord[3, 1]) / \
             (vertCoord[0, 0] - vertCoord[3, 0]) * nodeCoordX
             + vertCoord[3, 1]
nodeCoord[3*N:4*N, 0] = nodeCoordX[1:-1]
nodeCoord[3*N:4*N, 1] = nodeCoordY[1:-1]
nodeGrad[3*N:4*N, 0] = nodeCoordX[1:-1]
nodeGrad[3*N:4*N, 1] = (vertCoord[0, 1] - vertCoord[3, 1]) / \
                       (vertCoord[0, 0] - vertCoord[3, 0])
                       * np.ones(N)
nodeTangent[3*N:4*N, 0] = vertCoord[0, 0] - vertCoord[3, 0]
nodeTangent[3*N:4*N, 1] = vertCoord[0, 1] - vertCoord[3, 1]

# Define rotation in clockwise with respect to [0,0]
theta = theta * np.pi / 180
thetaDot = thetaDot * np.pi / 180
rMat = np.matrix([[np.cos(theta), -np.sin(theta)],
                  [np.sin(theta), np.cos(theta)]])
rMatDot = thetaDot * np.matrix([[-np.sin(theta), -np.cos(theta)],
                                [np.cos(theta), -np.sin(theta)]])
rMat90 = np.matrix([[np.cos(np.pi/2), -np.sin(np.pi/2)],
                    [np.sin(np.pi/2), np.cos(np.pi/2)]])

nodeCoordDot = np.zeros([4*N, 2])
for ni in range(0, nodeCoord.shape[0]):
    nodeCoord[ni, :] = (rMat * nodeCoord[ni, :].reshape([2, 1]))
                       .reshape([1, 2])
    nodeCoordDot[ni, :] = (rMatDot * nodeCoord[ni, :].reshape([2, 1]))
                          .reshape([1, 2])
    nodeGrad[ni, 1] = (nodeGrad[ni, 1] + np.tan(theta))
                      / (1 - nodeGrad[ni, 1] * np.tan(theta))
    nodeTangent[ni, :] = nodeTangent[ni, :]
                         / np.linalg.norm(nodeTangent[ni, :])
    nodeTangent[ni, :] = (rMat * nodeTangent[ni, :].reshape([2, 1]))
                         .reshape([1, 2])
    nodeNormal[ni, :] = (rMat90 * nodeTangent[ni, :].reshape([2, 1]))
                        .reshape([1, 2])


np.savetxt('coord.txt', nodeCoord, '%2.2f')
np.savetxt('coordDot.txt', nodeCoordDot, '%2.2f')
np.savetxt('grad.txt', nodeGrad, '%2.2f')
np.savetxt('normal.txt', nodeNormal, '%2.2f')
```

References

[1] Ashley, H., "Piston theory-a new aerodynamic tool for the aeroelastician," *Journal of the Aeronautical Sciences*, Vol. 23, No. 12, 1956, pp. 1109–1118.

[2] McNamara, J. J. and Friedmann, P. P., "Aeroelastic and aerothermoelastic analysis in hypersonic flow: past, present, and future," *Aiaa Journal*, Vol. 49, No. 6, 2011, pp. 1089–1122.