A Study on short period dynamics and stability of flexible missiles

Thesis submitted in partial fulfillment of the
requirement for the degree of

Bachelor of Technology
in
Mechanical Engineeting

Submitted by
Subhrajit Bhattacharya
Roll no.: 02ME1041

Under the supervision of
Prof. Anirvan Dasgupta
Department of Mechanical Engineering,
IIT Kharagpur

Department of Mechanical Engineering,
Indian Institute of Technology,
Kharagpur – 721302.

May, 2006.

Certificate

This is to certify that the report entitled "A Study on short period dynamics and stability of flexible missiles" submitted by Subhrajit Bhattacharya to the Department of Mechanical Engineering, IIT Kharagpur, is a bona fide record of work carried out under my supervision and guidance. This meets the requisites and standards as per regulation of the institute for being considered as thesis in partial fulfillment of the requirement for the degree of Bachelor of Technology (Hons.).

Prof. Anirvan Dasgupta,
Dept. of Mechanical Engineering,
IIT Kharagpur.

Date:

Acknowledgement

I would like to thank my project guide Prof. Anirvan Dasgupta, Department of Mechanical Engineering, IIT Kharagpur, and would also like to thank Prof. Siddhartha Mukhopadhyay, Department of Electrical Engineering, IIT Kharagpur, for their invaluable and experienced suggestions and whole-hearted help, without which this work would have not been possible. I would also like to thank Mr. Ranajit Das of Department of Electrical Engineering, IIT Kharagpur, for his extensive help and engrossed discussions. And finally I want to express my whole-hearted gratitude towards the Department of Mechanical Engineering and Indian Institute of Technology, Kharagpur for providing me this wonderful opportunity of pursuing my under-graduate studies in mechanical engineering and hence giving me the scope to learn from and work with the erudite faculties of this esteemed institute.

Contents

1 Introduction and literature review

1.1 Introduction to the problem

Missiles and rockets used in defense as well as in aerospace research and applications are generally structures which assume very high speed (typically above 3 Mach). But they are also generally very slender structure with a high L/d ratio in order to increase the aerodynamic efficiency. Hence it is very important to take into account the structural flexibility while studying the dynamics of the missile or designing a control system for it.

Real time trajectory control and stabilization of guided missiles is a challenging problem both in respect to study of dynamics of the system and designing of suitable control loops. There had been several works previously on missiles considering them as rigid structures [1, 2]. Study of the rigid body dynamics and design of control system has been done and implemented successfully. Such control loops involve velocity, position and orientation feedbacks along with actuation by thrust vectoring.
However at high speeds and high L/d ratios it becomes indispensable to consider the structural stability of the system and to ensure that the system stabilizes quickly in case of slight deviations from the expected trajectory/orientation/velocity. Such works have been investigated previously [1, 3, 5].
The control system for the whole missile may be viewed to be consisting of two loops, an external one for trajectory control and long period dynamics, and an internal one for stabilizing the perturbation components over a short period. It can be pointed out over here that in the outer loop we need not take into consideration the structural flexibility since the structural deformation is not a parameter that needs to be controlled for trajectory control. However in case of the internal loop, attention should be given so that the structural instabilities, if any, should die down quickly.

Another important cause of instabilities due to perturbations is the formation of turbulence in the flow past the missile. In order to ensure that the flow along the sides of the missile never reach the turbulent zone, we need to design the maximum length and the maximum velocity of the missile accordingly. In fact it can be shown that the stability of the flow depends on the Reynolds Number. However since the surface over which the flow is taking place is flexible, this becomes a problem of fluid-structure interaction.
The problem of fluid-structure interaction is encountered in various Engineering applications. In the present problem we have studies the vibrations induced into structures due to flow taking place on its surface, and hence analyzing the stability of the flow. Though the present approach to this part of the problem has been grossly simplified and no substantial simulation or numerical solutions could be made, the work have great potential in applications like flow over underwater vehicle, flow over aerospace structures, design of coating materials on the surface of bodies moving through fluids in order to reduce the turbulence, etc.

1.2 Literature Review

As said earlier, several works have been done on modeling and control of missiles, both with and without considering the structural flexibility.
Pourtakdoust and Assadian [3] have modeled the missile as Bernoulli–Euler beam and have derived the effect of axial and transverse components of engine thrust on the beam. Guran and Ossia [5] have done works on similar line with Galerkin method and Finite difference solution approaches.
T.P.Chang [4] in his work has also modeled space structures as free-free beam and has studies it under random excitation forces.
Greensite [1] in his book has given a detailed discussion on short period dynamics of flexible missiles. The present work is primarily based on the theories given in this book. The short period dynamics of a system is studied in order to ensure that the system stabilizes under some small perturbations in the external forces or state variables. C.T.Leondes [12] gives a detailed discussion on contril and guidance of aerospace vehicles.
The problem of studying stability of flows over rigid and flexible surfaces has been extensively reviewed by Betchov and Criminale [7]. Benjamin [8] has done works on similar line. The Automated Search for Eigenvalues used for eigenvalue solution of Orr-Sommerfeld equation was successfully used by Betchov and Szewczyk [10]. The solution using two integration passes was used by Nachtsheim [11].

1.3 Overview of the present work

The present work primarily deals with the short period dynamics of a missile, taking into account the structural flexibility. By short period dynamics we mean that over the period the desired motion (or the steady state motion) remains independent of time, while only the perturbation components are studied.
To approach the problem, perturbation components of the system variables have been taken and the dynamics of the system has been linearized. A control loop is to be designed in such a way that the perturbation components die down with time.
This part of the work mainly consists of the following parts:

I. Study and modeling of the system dynamics. These include
   a) The overall balancing of forces/moments on the system,
   b) Inclusion of forcing on the system due to thrust, engine inertia, aerodynamic forces, sloshing of fuel, gravity, etc.
   c) Considering structural flexibility and how it's being affected by the foresaid forces.

II. Linearizing the equations obtained in the above step by elimination of the steady state components of state variables, forces and moments. By steady state we mean the condition in which the missile is having a constant linear and angular velocity with no structural deformation/vibration. The forces and moments acting in the system under such a situation are the steady-state components of the forces and moments.

III. Implementing the linearized equations using Matlab 6.5 so as to create a virtual model of the system's short period dynamics. The implementation is

done using state-variables and matrix formulation so that it becomes easy to study the system. A few salient features of the implemented code are:

- Flexible and easily customizable code with separate functions for overall rigid-body equations, flexibility, forcing due to thrust, engine inertia, aerodynamic forces, etc. Hence it is possible to switch on or off one or more factors in the code very easily.
- State-space representation of the system equations enabled easy and fast update of the state vector and perform checks on stability of the system.
- The equations and expressions in the code are expressed as vectors with the elements being the coefficients of the state variables and their derivatives. This technique of implementation enabled keeping the state variables and their derivatives separate from each other in every equation or expression. Hence it helped in easy creation of the matrices in the state-space representation.

The concepts and related equations for part I and II have been extensively adopted from the text by Greensite [1]. However a few modifications along with change in some sign conventions have been done.

The next part consists of studying the stability of flow past the missile. The present work deals with analysis of the stability of 2wo-dimensional fluid flows over a flexible flat surface. By the term ‘stability’ we mean that we try to determine the critical Reynolds number for a given flow profile over the structure.
We will consider flexibility of the beam (since the missile has been modeled as a beam) and frame the corresponding equations. Our main aim has been numerical eigenvalue solution of the Orr- Sommerfeld equation with appropriate boundary conditions. For the purpose of numerical solution we have presently investigated only the case of flow over a rigid surface to check if we obtain the standard results for stability of parallel flow over rigid surfaces available in standard text.

2 Mathematical model of the flexible missile and design of Control Loop

2.1 The dynamic model

The model of the missile has been simplified by considering it as a flexible beam with the engine and fuel (modeled as harmonic pendulums) attached to it. Hence the basic model includes a free-free flexible beam with forcing on it due to thrust, engine inertia, aerodynamics, sloshing of fuel, gravity, etc.
The following free body diagram gives an overall representation of the system.

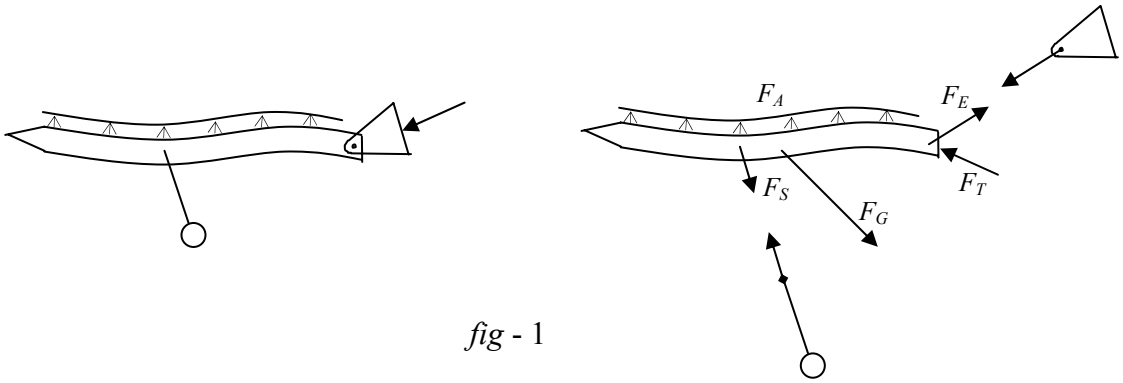


fig - 1

2.2 Coordinate system and notations

A local right-handed coordinate system (X_b, Y_b, Z_b) is fixed to the missile with its origin chosen at any point on its body. It's called the body frame, S_b. The X-axis is chosen along the direction of the instantaneous velocity's mean component (i.e. removing the perturbation components). The Z-axis is chosen perpendicular to the X-axis on the plane of instantaneous radius of curvature.

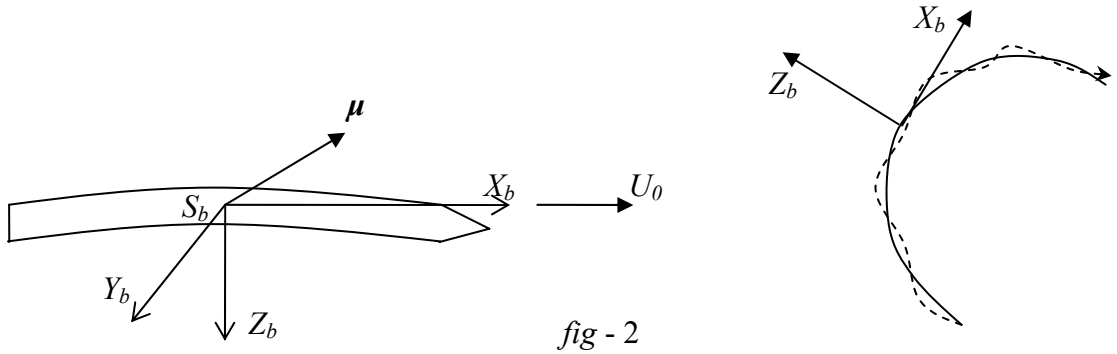


fig - 2

Notations:

L_0 = Net length of the missile body;
L_C = Distance of the origin of S_b from hind end of the missile;
L_R = Dist between c.g. of engine and the hinge to which the engine is attached with the missile body;
L_A = Distance between origin of S_b and nose tip of vehicle = $L_0 - L_C$;
m_R = Mass of engine; m_M = Mass of missile without engine; $m_0 = m_M + m_R$ = Total mass.

2.3 Overview of equations governing the motion of the missile

The equations are to be developed in 3 steps:

i. Overall equation of motion for net linear and angular velocities considering the net forces and moments on the beam.
ii. Considering flexibility of the beam and the forcing on it, determination of the shape of the beam.
iii. Determination of the forces due to the various factors like thrust, engine inertia, aerodynamics, sloshing of fuel, gravity, etc.

In the next section equations for each of these will be gradually developed with mention of appropriate assumptions and then the equations will be linearized using perturbations in each variable.

2.4 Overall equation for rigid body dynamics

2.4.1 Force Equations

The net acceleration of the missile body (excluding engine & fuel) in S_b is given by,

$$\mathbf{a} = \frac{d\boldsymbol{\mu}}{dt} = \frac{\partial \boldsymbol{\mu}}{\partial t} + \boldsymbol{\omega} \times \boldsymbol{\mu} + \boldsymbol{\omega} \times \left(\boldsymbol{\omega} \times \boldsymbol{\rho}_c \right) + \frac{\partial \boldsymbol{\omega}}{\partial t} \times \boldsymbol{\rho}_c \tag{1}$$

Where,
$\boldsymbol{\rho_c}$ is position of the centre of mass of the missile body in S_b,
$\boldsymbol{\mu}$ is the velocity of origin of S_b,
$\boldsymbol{\omega}$ is the angular velocity of the frame S_b relative to the global inertial frame,
and the partial derivatives are taken assuming that S_b is not rotating, i.e. the unit vectors are constant.

Now from Newton's Second Law of motion,

$$\boldsymbol{F} = m_M \boldsymbol{a}$$

where, $\boldsymbol{F}$ is the net external force on the missile body.

Now putting,

$$\begin{aligned} \boldsymbol{\mu} &= U\mathbf{i}_b + V\mathbf{j}_b + W\mathbf{k}_b \\ \boldsymbol{\omega} &= P\mathbf{i}_b + Q\mathbf{j}_b + R\mathbf{k}_b \\ \boldsymbol{\rho}_c &= x_{cg}\mathbf{i}_b + y_{cg}\mathbf{j}_b + z_{cg}\mathbf{k}_b \end{aligned} \tag{2}$$

we get,

$$\begin{aligned} \mathbf{F} = & \left[m_M \left(\dot{U} + QW - RV \right) - m_M x_{cg} \left(R^2 + Q^2 \right) - m_M y_{cg} \left(\dot{R} - PQ \right) + m_M z_{cg} \left(\dot{Q} + PR \right) \right] \mathbf{i}_b \\ & + \left[m_M \left(\dot{V} + RU - PW \right) + m_M x_{cg} \left(\dot{R} + PQ \right) - m_M y_{cg} \left(R^2 + P^2 \right) - m_M z_{cg} \left(\dot{P} + QR \right) \right] \mathbf{j}_b \\ & + \left[m_M \left(\dot{W} + PV - QU \right) - m_M x_{cg} \left(\dot{Q} - PR \right) + m_M y_{cg} \left(\dot{P} + QR \right) - m_M z_{cg} \left(Q^2 + P^2 \right) \right] \mathbf{k}_b \end{aligned} \tag{3}$$

2.4.2 Moment equations

The net external moment due to the forces and couples in the body frame S_b can be expressed as,

$$\mathbf{G}_b = \frac{d\mathbf{H}_b}{dt} + m_M \boldsymbol{\rho}_c \times \frac{d\boldsymbol{\mu}}{dt} \tag{4}$$

where,
$\boldsymbol{H}_b$ is the net angular momentum of the missile in the body coordinate frame, given by,

$$\mathbf{H}_b = H_X \mathbf{i}_b + H_Y \mathbf{j}_b + H_Z \mathbf{k}_b$$

where,

$$\begin{aligned} H_X &= I_{XX}P - I_{XY}Q - I_{XZ}R \\ H_Y &= -I_{XY}P + I_{YY}Q - I_{YZ}R \\ H_Z &= -I_{XZ}P - I_{YZ}Q + I_{ZZ}R \end{aligned} \tag{5}$$

where I_{XX}, I_{XY}, I_{ZX}, I_{YY}, I_{YZ}, I_{ZZ} are the second moments of inertia of the missile body in S_b.
And,

$$\frac{d\mathbf{H}_b}{dt} = \frac{\partial \mathbf{H}_b}{\partial t} + \boldsymbol{\omega} \times \mathbf{H}_b \tag{6}$$

Using (4), (5) and (6) we obtain,

$$\begin{aligned} \mathbf{G}_b = &[I_{XX}\dot{P} - I_{XY}(\dot{Q} - PR) - I_{XZ}(\dot{R} + PQ) + I_{YZ}(R^2 - Q^2) + (I_{ZZ} - I_{YY})QR \\ &+ m_M y_{cg}(\dot{W} + PV - QU) - m_M z_{cg}(\dot{V} + RU - PW)]\mathbf{i}_b \\ &+ [-I_{XY}(\dot{P} + QR) + I_{YY}\dot{Q} - I_{YZ}(\dot{R} - PQ) + I_{XZ}(P^2 - R^2) + (I_{XX} - I_{ZZ})PR \\ &- m_M x_{cg}(\dot{W} + PV - QU) + m_M z_{cg}(\dot{U} + QW - RV)]\mathbf{j}_b \\ &+ [-I_{XZ}(\dot{P} - QR) - I_{YZ}(\dot{Q} + PR) + I_{ZZ}\dot{R} + I_{XY}(Q^2 - P^2) + (I_{YY} - I_{XX})PQ \\ &+ m_M x_{cg}(\dot{V} + RU - PW) - m_M y_{cg}(\dot{U} + QW - RV)]\mathbf{k}_b \end{aligned} \tag{7}$$

Now we linearize the equations (3) and (7) by expressing U, V, W, P, Q and R as summation of steady-state and perturbation components:

$$\begin{aligned} U &= U_0 + u & P &= P_0 + p \\ V &= V_0 + v & Q &= Q_0 + q \\ W &= W_0 + w & R &= R_0 + r \end{aligned} \tag{8}$$

where, u, v, w, p, q and r are the perturbation components and are small compared to the steady state components. Moreover, because of the choice of our body frame S_b, we may assume that the primary steady-state component of velocity is only along $\boldsymbol{i}_b$. And we also assume that in steady state, for studying the short period dynamics, the desired trajectory is a straight line. Hence the steady state components of angular velocity are all zero.
Hence we have,

$$\begin{aligned} V_0 = 0,\ W_0 = 0 \\ P_0 = 0,\ Q_0 = 0,\ R_0 = 0. \end{aligned} \tag{9}$$

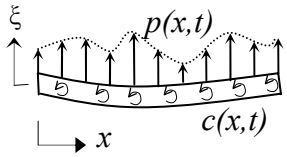
Making these substitutions, eliminating the steady-state components and simplifying the results, we obtain the following sets of equation:

$$
\begin{aligned}
F_X &= m_M(\dot{u} - y_{cg}\dot{r} + z_{cg}\dot{q}) \\
F_Y &= m_M(\dot{v} + U_0 r + x_{cg}\dot{r} - z_{cg}\dot{p}) \\
F_Z &= m_M(\dot{w} - U_0 q - x_{cg}\dot{q} + y_{cg}\dot{p}) \\
M_X &= I_{XX}\dot{p} - I_{XY}\dot{q} - I_{XZ}\dot{r} + m_M y_{cg}(\dot{w} - U_0 q) - m_M z_{cg}(\dot{v} + U_0 r) \\
M_Y &= -I_{XY}\dot{p} + I_{YY}\dot{q} - I_{YZ}\dot{r} - m_M x_{cg}(\dot{w} - U_0 q) + m_M z_{cg}\dot{u} \\
M_Z &= -I_{XZ}\dot{p} - I_{YZ}\dot{q} + I_{ZZ}\dot{r} + m_M x_{cg}(\dot{v} + U_0 r) - m_M y_{cg}\dot{u}
\end{aligned}
\qquad (10)
$$

Where F_X, F_Y, F_Z, M_X, M_Y and M_Z are perturbation components of the forces and moments. It can be mentioned here that the 6 equations in (10) gives the dynamics of the perturbation components of linear and angular velocities.

2.5 Equations for elastic vibration of the missile body

Though the deflection of the missile body is possible in both the pitch (i.e. X_bZ_b) and yaw (i.e. X_bY_b) planes, we have presently restricted our analysis to pitch plane only assuming that the major aerodynamic forces, the thrust force and other forces have negligible components along Y_b-direction.
The missile is considered to be a free-free beam under external loadings and moments. It's assumed that the beam obeys the Euler Lagrangian beam equation:



b.c. for free-free beam:

$\xi''(0,t)=0$ $\quad$ $\xi''(L_0,t)=0$
$\xi'''(0,t)=0$ $\quad$ $\xi'''(L_0,t)=0$

fig - 3

The equation governing the deflection of the beam, $\xi(x, t)$, is given by,

$$\rho A\ddot{\xi} + EI\frac{\partial^4 \xi}{\partial x^4} = p(x,t) + \frac{\partial c(x,t)}{\partial x}$$

where, p and c are distributed load and couple per unit length on the beam. It can me mentioned here that since there is no deflection in steady state, p and c are due to only the perturbation components of forces and moments.
ρ, A, E and I are the mass density, cross-sectional area, Young's modulus and second moment of area of the cross section about the neutral axis respectively.

2.5.1 Determining the natural frequencies and mode shapes

Now we seek eigen-value solution for the beam equation. For this we remove the forcing terms and assume $\xi(x, t)$ is of the form, $\phi(x)e^{i\omega t}$. The resulting non-trivial solutions that satisfy the boundary conditions will give the eigen-values ω and the corresponding eigenfunctions (i.e. mode shapes) ϕ.

The equation to be solved for obtaining ϕ is,

$$\frac{\partial^4 \phi}{\partial x^4} - \beta^4 \phi = 0\,, \tag{11}$$

where,

$$\beta = \left(\frac{\omega^2 \rho A}{EI} \right)^{1/4}. \tag{12}$$

Hence, the solution is of the form,

$$\phi(x) = A\cosh \beta x + B\sinh \beta x + C\cos \beta x + D\sin \beta x \tag{13}$$

Substituting this ϕ in the four boundary conditions, i.e.,

$$\phi''(0) = 0,\ \phi'''(0) = 0,\ \phi''(L_0) = 0,\ \phi'''(L_0) = 0$$

we obtain,

$$\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ \cosh \beta L_0 & \sinh \beta L_0 & -\cos \beta L_0 & -\sin \beta L_0 \\ \sinh \beta L_0 & \cosh \beta L_0 & \sin \beta L_0 & \cos \beta L_0 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = 0 \tag{14}$$

Eliminating A, B, C and D, we obtain the characteristic equation,

$$\cos \beta L_0 \cdot \cos \beta L_0 - 1 = 0 \tag{15}$$

This equation is solved for β numerically.

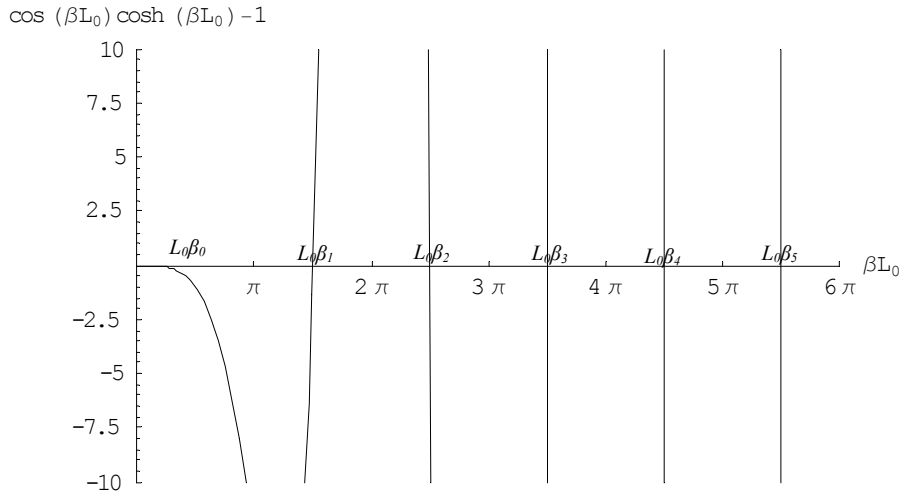


fig - 4

The above plot shows the solution points.
It can be noted that $L_0\beta_n \approx (n + 1/2)\pi$ for $n \geq 3$. We used this approximation for calculating β_n for $n \geq 4$.

Once β_n is know, we obtain ω_n from (13).

From (14), $A_n=C_n$, $B_n=D_n$.
Putting $A_n = -\left(\sinh(\beta_n L_0) - \sin(\beta_n L_0)\right)$, we obtain $B_n = \cosh(\beta_n L_0) - \cos(\beta_n L_0)$.

Hence, from (13), the n[th] mode shape is obtained as,

$$\begin{aligned}\phi_n(x) = &-\left(\sinh(\beta_n L_0) - \sin(\beta_n L_0)\right)\left(\cosh(\beta_n x) + \cos(\beta_n x)\right) \\ &+\left(\cosh(\beta_n L_0) - \cos(\beta_n L_0)\right)\left(\sinh(\beta_n x) + \sin(\beta_n x)\right)\end{aligned} \tag{16}$$

2.5.2 Discretization of governing equation of dynamics of the beam

Now the shape of the beam can be expressed as a linear superposition of all the mode shapes. i.e.,

$$\xi(x,t) = \sum_{n=1}^{\infty} q_n(t)\,\phi_n(x) \tag{17}$$

Substituting (17) in (11) and using (12) we have,

$$\mu(x)\sum_{n=1}^{\infty}\left(\ddot{q}_n + \omega_n^2 q_n\right)\phi_n(x) = p(x,t) + \frac{\partial c(x,t)}{\partial x} \tag{18}$$

Where, $\mu(x)$ is the mass per unit length of the missile = ρA.

We multiply (18) by $\phi_i(x)$ on either sides and integrate from $x = 0$ to L_0. We note that the mode shapes are orthogonal to each other, which makes all the other terms except the one with $\phi_i(x)$ vanish. Hence we have,

$$\ddot{q}_i + \omega_i^2 q_i = \frac{Q_i}{M_i} \tag{19}$$

where,

$Q_i(t) = \int_0^{L_0}\left(p(x,t) + \frac{\partial c(x,t)}{\partial x}\right)\phi_i(x)dx$ is the generalized force,

and $M_i(t) = \int_0^{L_0}\mu(x)\left[\phi_i(x)\right]^2 dx$ is the generalized mass.

Assuming the presence of viscous damping in the system, (19) is modified as,

$$\ddot{q}_i + 2\zeta_i\omega_i\dot{q}_i + \omega_i^2 q_i = \frac{Q_i}{M_i} \tag{20}$$

Equation (20) is to be solved for q_i, i = 1 to ∞. However for simulation purpose, only first 10 modes were taken, i.e. i = 1 to 10.

It can be again noted here that all the calculations that have been performed are restricted to the pitch plane, i.e. the Z_bX_b plane. However in similar fashion the model may be extended to the yaw plane without much problem.

2.6 Perturbation components of forces and moments on the flexible missile body

Now that the basic equations governing the perturbation motion of the flexible missile body have been developed [i.e. eqn (10) and (20)], we need to determine the distributed forces and couples, *p* & *c* and the perturbation forces and moments. The primary contributing factors to these terms are thrust, engine inertia, aerodynamics, sloshing of fuel and gravity. In the following section the contribution due to each of these are discussed in brief. The detailed derivations are given in Greensite [1]. Since at present we have restricted our model to only the pitch plane, we are required to find the components of forces only in the X_b and Z_b directions, and moments along Y_b.

2.6.1 Forces and moments due to Thrust from the engine (F_T & M_T)

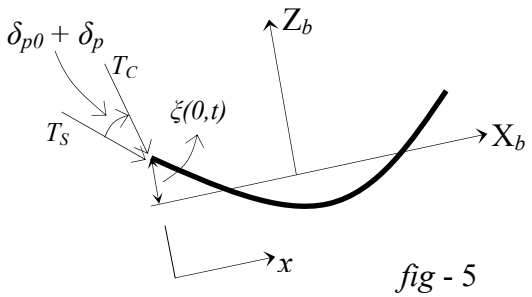


fig - 5

It may be noted in the adjacent diagram that there are two components of the thrust, namely T_S and T_C. Both of them are provided by the engine. T_S acts along the tangent at the hind end of the missile, while T_C acts along an angle $\delta_{p0}+ \delta_p$ from the tangent direction. The angle δ_p is the perturbation component of the engine swivel angle.

By direct resolution of forces and moments, and elimination of steady state components, the perturbation components of forces acting at the hind end of the missile due to thrust are,

$$F_{T,X} = 0 \tag{21a}$$

$$F_{T,z} = T_C\delta_p - (T_C + T_S)\sum_i q_i(t)\,\phi_i'(0) \tag{21b}$$

And perturbation component of moment in X_bZ_b plane,

$$M_T = L_C\left[T_C\delta_p - (T_C + T_S)\sum_i q_i(t)\,\phi_i'(0)\right] - (T_C + T_S)\sum_i q_i(t)\,\phi_i(0) \tag{22}$$

It can be noted that the force is acting at a concentrated point at the hind end of the missile body. In order to incorporate this into the generalized force of eqn (20), we define the contribution of this force in the distributed force per unit length *p(x,t)* as,

$$p_T(x,t) = F_T\delta(x)$$

and the distributed couple, $c_T(x,t) = 0$ (23)

where, $\delta(x)$ represents the Dirac-delta function.

2.6.2 Forces and moments due to Inertia of the engine (F_E & M_E) and the swiveling moment

The expression for perturbation component of force acting at the hind end of the missile has been derived by extensively calculating the Kinetic energy of the engine and the using it in Lagrange's equation [1]. However the calculations have been re-done in order to consider our different sign convention for angles and ξ.

The final forces due to engine inertia acting at the hind end of the missile are given by,

$$F_{E,X} = -m_R \dot{u} \tag{24a}$$

$$F_{E,Z} = m_R \left[-L_R \ddot{\delta}_p - (L_C + L_R)\ddot{\theta} - \dot{w} - \sum_i \left[\phi_i(0) - L_R \phi_i'(0) \right] \ddot{q}_i(t) \right] \tag{24b}$$

And the couple due to the engine swivel is given by,

$$C_E = m_R L_R \left[\dot{w} + L_C \dot{q} + \sum_i \ddot{q}_i(t)\, \phi_i(0) \right] + I_R \left(\ddot{\delta}_p + \dot{q} - \sum_i \ddot{q}_i(t)\, \phi_i'(0) \right) \tag{25}$$

Hence, the net moment is given by,

$$M_E = L_C F_{E,Z} + C_E \tag{26}$$

Again as before, we take contribution of forces and couples due to engine inertia and swiveling into the distributed force and couple per unit length as,

$$p_E(x,t) = F_E \delta(x)$$

and,

$$c_E(x,t) = C_E \delta(x) \tag{27}$$

where, $\delta(x)$ represents the Dirac-delta function.

2.6.3 Forces and moments due to Aerodynamic forces (F_E)

Aerodynamic forces in the pitch plane act as a distributed force along the missile body, varying according to the angle of attack at that position.
The force per unit length along the Z_b axis at a position where the pitch plane angle of attack (i.e. angle between the relative velocity between the missile body at that position & the air and the tangent along the missile body at that location) is α' is given by,

$$p_A(x) = \frac{1}{2} \rho_{air} U^2 A_3 \frac{\partial C_N}{\partial \alpha} \alpha' \tag{28}$$

where, C_N = lift coefficient and is a function of angle of attack α, and may be position. It is assumed that C_N is proportional and linear to α for small values of α. Hence, $\frac{\partial C_N}{\partial \alpha}$ is assumed to be constant.

A_3 = a constant and is called the reference area used in calculating C_N. It's assumed to be 1.

An expression for α' can be determined from the angle between the relative velocity and the tangent to the body of the missile at the position of interest.

$$\alpha' = -\left(\alpha + \frac{q}{U}(L_A - L_0 + x) + \frac{\partial \xi}{\partial x} + \frac{1}{U}\frac{\partial \dot{\xi}}{\partial x} \right) \tag{29}$$

Substituting this α' in (28) we get,

$$p_A(x) = -\frac{1}{2}\rho_{air} U^2 A_3 \frac{\partial C_N}{\partial \alpha}\left[\alpha + \frac{q}{U}(L_A - L_0 + x) + \sum_i q_i(t)\,\phi_i'(x) + \frac{1}{U}\sum_i \dot{q}_i(t)\,\phi_i(x) \right] \tag{30}$$

Here we note U = U0 + u.

And the moment due to this distributed force is given by,

$$\left(x - L_A\right) p_A(x) \tag{31}$$

2.6.4 Forces and moments due to Gravity (F_G and M_G)

The Euler Angles relating the orientation of the global coordinate frame S_0 and the local coordinate frame S_b be $\psi_0+\psi$, $\theta_0+\theta$ and $\varphi_0+\varphi$. Hence S_0 is transformed to S_b by rotation by these angles about Z_0, Y_0 and X_0 respectively. Here ψ_0, θ_0 and φ_0 denote the steady state components and ψ, θ and φ are the perturbation components.

It may be proved that $p = \dot{\varphi}, \quad q = \dot{\theta}, \quad r = \dot{\psi}$.

Assuming the gravitational force acts along $-Z_0$, the perturbation components of forces and moment due to gravity on the pitch plane are given by,

$$F_{G,X} = m_0 g\theta \sin\theta_0 \tag{32a}$$

$$F_{G,Z} = -m_0 g\theta \cos\theta_0 \tag{32b}$$

$$M_G = m_0\left(z_{cg}\theta \sin\theta_0 + x_{cg}\theta \cos\theta_0\right) \tag{33}$$

These forces and moments are calculated about the origin of S_b. and assumed to be acting at that point. Hence we again define,

$$p_G(x,t) = F_{G,Z}\,\delta\left(x - (x_{cg} + L_C)\right) \tag{34}$$

We note that here we have to use ψ, θ and φ which are the integrals of p, q and r.

2.6.5 Force and Moment due to Sloshing of Fuel (F_S and M_S)

The presence of liquid fuel adds some extra degrees of freedom to the system. The liquid fuel is modeled as combination of several simple harmonic pendulum attached to the missile body. This is called the "hydrodynamic analogy" [9].

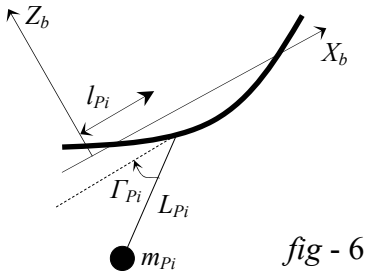


fig - 6

As shown in the adjacent figure, the i^{th} pendulum has been attached to the missile body at a distance l_{Pi} from origin of S_b.
The dynamics of a single pendulum can be found using Lagrange's equation of motion.

The dynamics of the i^{th} pendulum is governed by the following equation:

$$\ddot{\Gamma}_{Pi} + \frac{\dot{U}}{L_{Pi}}\Gamma_{Pi} = \frac{1}{L_{Pi}}\left[U_0 - \dot{w} + \dot{q}\left(l_{Pi} - L_{Pi}\right) + \sum_i \ddot{q}_i(t)\,\phi_i(l_{Pi})\right] \tag{35}$$

And the resulting forces and moments are hence given by,

$$F_{S,X} \simeq 0 \tag{36a}$$

$$F_{S,Z} = \sum_i m_{Pi}\dot{U}\Gamma_{Pi} \tag{36b}$$

$$M_S = -\sum_i m_{Pi} l_{Pi}\dot{U}\Gamma_{Pi} \tag{37}$$

And as before,

$$p_S(x,t) = \sum_i m_{Pi}\dot{U}\Gamma_{Pi}\delta\left(x - (l_{Pi} + L_C)\right) \tag{38}$$

2.7 State space formulation of the system and design of gain matrix for state feedback

Now that we have obtained the governing equations (10), (20) and (35) of the system, and have obtained expressions for the perturbation components of the forces and moments to be used in equation (10) and (20), we now nee to identify the state variables.
We note that because of elimination of the steady state components the equations and the expressions for firces and moments should be linear in the state variables. Hence on substituting the expression for forces and moments we can express the equations (10) and (20) as linear in the state variables.

2.7.1 Identifying the state variables and the governing equations

The state variables can be identified to be,
u, v, w,
$\psi, \theta, \varphi,$
$p(=\dot{\varphi}), q(=\dot{\theta}), r(=\dot{\psi})$,
$q_i, r_i(=\dot{q}_i)$, [i = 1 to N_{modes}]
$\Gamma_{Pi}, \Phi_{Pi}(=\dot{\Gamma}_{Pi})$, [$i$ = 1 to N_{slosh}]

Hence there are [9 + 2 N_{modes} + 2 N_{slosh}] state variables.
Equation (10) gives 6 equations; (20) gives N_{modes} equations; (35) gives N_{slosh} equations;
$p = \dot{\varphi}, q = \dot{\theta}, r = \dot{\psi}$ gives 6 equations; $r_i = \dot{q}_i$ gives another N_{modes} equations;
$\Phi_{Pi} = \dot{\Gamma}_{Pi}$ gives another N_{slosh} equations.
Hence we have a total of [9 + 2 N_{modes} + 2 N_{slosh}] equations.
Let, N_S = 9 + 2 N_{modes} + 2 N_{slosh}

2.7.2 The state-space formulation

Let the state vector be represented by,

$$\mathbf{s} = \left[u\ v\ w\ p\ q\ r\ \varphi\ \theta\ \psi\ \ q_1\ q_2 \cdots q_{N_{modes}}\ \ r_1\ r_2 \cdots r_{N_{modes}}\ \ \Phi_1\ \Phi_2 \cdots \Phi_{N_{modes}}\ \ \Gamma_1\ \Gamma_2 \cdots \Gamma_{N_{slosh}}\ \right]^T \quad (39)$$

Let the input/control parameters be represented bu the vector **u**. The elements in **u** are such that their coefficients in the equations (10), (20) and (35) are constant. One parameter obeying that property is δ_p. Hence in the present problem, we choose u to be a single element vector: **u** = δ_p. We may note that the equations also contain terms in $\ddot{\delta}_p$ contributed by force and moments due to engine inertia and swivel.

Let the N_S equations when expanded and simplifies be represented in state-space form as,

$$\mathbf{A}\dot{\mathbf{s}} = \mathbf{B}\mathbf{s} + \mathbf{C_0}\mathbf{u} + \mathbf{C_1}\dot{\mathbf{u}} + \mathbf{C_2}\ddot{\mathbf{u}} + \mathbf{D} \quad (40)$$

Once the matrices in (40) are known, simulating the system by integration of the equation won't be difficult. We have used first order Runge-Kutta method to integrate this equation.
So our primary aim is now to determine **A**, **B**, $\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$ and **D** from equations (10), (20) and (35).

2.7.3 Stability of the system without feedback

Equation (40) may be re-written as,

$$\dot{\mathbf{s}} = \mathbf{A}^{-1}\left(\mathbf{Bs} + \mathbf{C_0u} + \mathbf{C_1}\dot{\mathbf{u}} + \mathbf{C_2}\ddot{\mathbf{u}} + \mathbf{D}\right)$$

Hence, if **u** is bounded and independent of **s**, the system will be stable iff all the eigenvalues of $\mathbf{A^{-1}B}$ have non-positive real parts.

2.7.4 Determination of gain in state feedback

2.7.4.1 Proportional Control

We take $\mathbf{u} = \mathbf{K}^T\mathbf{s}$, where **K** is the gain matrix. Since in the present problem, **u** has a single element, **K** is a vector with N_S elements.

Hence (40) becomes,

$$\mathbf{A}\dot{\mathbf{s}} = \mathbf{Bs} + \mathbf{C_0}\mathbf{K}^T\mathbf{s} + \mathbf{C_1}\mathbf{K}^T\dot{\mathbf{s}} + \mathbf{C_2}\mathbf{K}^T\ddot{\mathbf{s}} + \mathbf{D} \tag{41}$$

We substitute $\dot{\mathbf{s}} = \boldsymbol{\tau}$ and write $\boldsymbol{\varsigma} = \begin{bmatrix} \boldsymbol{\tau} \\ \mathbf{s} \end{bmatrix}$.

Hence, we have,

$$\dot{\boldsymbol{\varsigma}} = \begin{bmatrix} \left(\mathbf{C_2K^T}\right)^{-1}\left(\mathbf{A} - \mathbf{C_1K^T}\right) & -\left(\mathbf{C_2K^T}\right)^{-1}\left(\mathbf{B} + \mathbf{C_0K^T}\right) \\ \mathbf{I} & \mathbf{0} \end{bmatrix}\boldsymbol{\varsigma} \tag{42}$$

However the above equation's validity is subjected to the existence of $\left(\mathbf{C_2K^T}\right)^{-1}$. And as a matter of fact, this inverse won't exist since both $\mathbf{C}_2$ and **K** are vectors and hence the matrix $\mathbf{C_2K^T}$ is if rank 1. Hence we can't use proportional control.

2.7.4.2 Integral Control

Now we assume $\mathbf{u} = \mathbf{K}^T\int_0^t \mathbf{s}dt$.

This gives $\dot{\mathbf{u}} = \mathbf{K}^T\mathbf{s}$ and $\ddot{\mathbf{u}} = \mathbf{K}^T\dot{\mathbf{s}}$.

Substituting in (40),

$$\mathbf{A}\dot{\mathbf{s}} = \mathbf{Bs} + \mathbf{C_0K^T}\int_0^t \mathbf{s}dt + \mathbf{C_1K^T}\mathbf{s} + \mathbf{C_2K^T}\dot{\mathbf{s}} + \mathbf{D} \tag{43}$$

Substituting $\int_0^t \mathbf{s}dt = \boldsymbol{\tau}$ i.e., $\dot{\boldsymbol{\tau}} = \mathbf{s}$, and writing $\boldsymbol{\varsigma} = \begin{bmatrix} \mathbf{s} \\ \boldsymbol{\tau} \end{bmatrix}$,

We have,

$$\dot{\boldsymbol{\varsigma}} = \begin{bmatrix} \left(\mathbf{A} - \mathbf{C}_2\mathbf{K}^{\mathrm{T}}\right)^{-1}\left(\mathbf{B} + \mathbf{C}_1\mathbf{K}^{\mathrm{T}}\right) & \left(\mathbf{A} - \mathbf{C}_2\mathbf{K}^{\mathrm{T}}\right)^{-1}\mathbf{C}_0\mathbf{K}^{\mathrm{T}} \\ \mathbf{I} & \mathbf{0} \end{bmatrix} \boldsymbol{\varsigma} \tag{44}$$

Now, since the inverse of A is expected to exist (else eqn (40) can't be integrated at all), and the rank of $\mathbf{C}_2\mathbf{K}^{\mathrm{T}}$ is 1, the matrix $\mathbf{A} - \mathbf{C}_2\mathbf{K}^{\mathrm{T}}$ is expected to be invertible.

Hence using integral control equation (44) is the primary equation to be solved in order to obtain the state at any instant of time.

2.7.5 Stability of the system with integral state feedback

Stability id determined by whether the eigenvalues of

$$\mathbf{M}(\mathbf{K}) = \begin{bmatrix} \left(\mathbf{A} - \mathbf{C}_2\mathbf{K}^{\mathrm{T}}\right)^{-1}\left(\mathbf{B} + \mathbf{C}_1\mathbf{K}^{\mathrm{T}}\right) & \left(\mathbf{A} - \mathbf{C}_2\mathbf{K}^{\mathrm{T}}\right)^{-1}\mathbf{C}_0\mathbf{K}^{\mathrm{T}} \\ \mathbf{I} & \mathbf{0} \end{bmatrix} \tag{45}$$

have non-positive real parts.

Out aim in designing **K** will be to place the poles of **M** to the left side of the imaginary axis. However achieving this analytically seems to be rather difficult. Hence we tried to adopt some numerical methods, though without much success!

2.7.6 Numerical methods for placing poles of M

For some particulat values of thrust and U_0, the open loop system was found to become unstable, i.e. some of the poles of **M** had positive real parts for **K** = 0. Hence we now try to design a suitable **K** for stabilizing the system.

2.7.6.1 Newton-Raphson's method for pole placement of **M**

For a given set of target poles $\lambda_1, \lambda_2, \ldots, \lambda_{2\mathrm{Ns}}$ for the matrix **M**, we define a vector function,

$$\mathbf{g}(\mathbf{K}) = \begin{bmatrix} \det[\mathbf{M}(\mathbf{K}) - \lambda_1\mathbf{I}] \\ \det[\mathbf{M}(\mathbf{K}) - \lambda_2\mathbf{I}] \\ \vdots \\ \det[\mathbf{M}(\mathbf{K}) - \lambda_{2N_S}\mathbf{I}] \end{bmatrix} \tag{46}$$

Hence we should find a **K** such that **g**(**K**)=0.

This was attempted by Newton-Raphson iteration for vector functions given by,

$$\mathbf{K}_{n+1}^{T} = \mathbf{K}_{n}^{T} - [\mathbf{g}(\mathbf{K}_n)]^{T}\left\{\nabla[\mathbf{g}(\mathbf{K}_n)]^{T}\right\}^{-1} \tag{47}$$

However this iteration failed to converge!

2.7.6.2 Genetic algorithm for searching a suitable K

We tried to search for a **K** such that the poles of **M** fall on the left side of imaginary line by using the principles of genetic algorithm. We started with an initial population of vectors **K** and determined fitness of each of them. The fitness function was designed so that more the eigenvalues are to the left of the complex plane, the better is the fitness. We performed crossing between the **K**'s with higher fitness and next continued with the new population until a **K** is obtained for which all the eigen values of **M** fall to the left side of the complex plane. However this method also failed to give a feasible solution.

2.7.6.3 Search for elements of K over a wide range using method of score assignment

In this method we start with an initial guess of **K**. Then we take each element of **K** at a time and cause it to change and hence observe how the eigenvalues are changing. Say we are considering K_i.
We define a 'score' s_i which is defined as follows:

$$s_i(\Delta) = c\sum\left[\mathrm{Re}\left\{eig_i[M(K)]\right\} - \mathrm{Re}\left\{eig_i[M(K^{C_i}(\Delta))]\right\}\right] \; + \; n^{-}\Theta \; - \; n^{+}\Theta \qquad (48)$$

where $K^{C_i}(\Delta)$ represents the K matrix when its i^{th} element is changed by Δ,
eig_i represents the i^{th} eigenvalue,
n^- is the number of eigenvalues that have moved from the positive real to negative real side of the complex plane because of change of K to $K^{C_i}(\Delta)$,
and n^+ is the number of eigenvalues that have moved from the negetive real to positive real side of the complex plane because of change of K to $K^{C_i}(\Delta)$,
c is a constant.

Evidently we will prefer a change for which the score is high. Hence we find the changes, Δ, for each K_i for which $s_i(\Delta)$ is maximized. However $s_i(\Delta)$ was found to be changing very arbitrarily with Δ. Hence Newton-Raphson iteration for this optimization didn't succeed.

The search for Δ was performed over a wide range by the method of *bracketing*. In this method a wide zone of Δ was chosen and it was divided into a finite number of sub-zones. The value of $s_i(\Delta)$ is then evaluated at the mid-point of each sub-zone. The zone for which the value is maximum is retained and the remaining discarded. This new zone is then again divided and the process is continued for several number of steps.

Finally only the top 2-3 K_i's are chosen and the corresponding optimized changes are added to the respective elements of **K**. This process is repeated unless a **K** is obtained corresponding to which all the eigenvalues of **M** lie to the left side of the imaginary axis.

This method worked and could finally give a **K** for which all the poles of **M** were placed to the left side of the imaginary axis.
However a few elements of **K** hence obtained were extremely large and resulted in eigenvalues of very large magnitude. An attempt for solving eqn (44) using this **K** led

to divergence. However it was found that if the time steps for integration of (44) could be decreased to a great extent, no divergence was detected. However this also meant slow calculation and even after running the program with this reduced time steps no observable changes in the state variables was observed. Hence, though a mathematically feasible solution fore **K** could be obtained, it's practical feasibility in either simulations or real model is questionable!

3 Numerical simulations, results and discussions

3.1 Implementation in MATLAB code

In the present implemented model, everything except force due to sloshing has been incorporated. As mentioned before, the model has been restricted to pitch plane. However the code is kept flexible enough in order to incorporate these things very easily. The primary features of the code are:

State vector representation: The state vector is defined as in (39) except the last 2 N_{slosh} elements for the sloshing.

Expressions defined as vectors: It is observed from eqn (10) and (20) that if we write the forces and moments in the equation and try to simplify them in pen-paper, the equations hence obtained will not only become huge and cumbersome, but also difficult to debug or modify. Hence we adopted an innovative way of representing the expressions for Force, Moment, etc. and then use them directly in equation (10) and (20), while keeping the individual terms of the state variables and their derivatives separate from each other.

We represent an expression in form of a vector where the elements represent the coefficients of the state variables and their derivatives, and the other constant terms.

Hence, if we define,

$$\mathbf{v} = \left[1 \; \ddot{\mathbf{u}} \; \dot{\mathbf{u}} \; \mathbf{u} \; \dot{u} \; \dot{v} \; \dot{w} \; u \; v \; w \; \dot{p} \; \dot{q} \; \dot{r} \; p \; q \; r \; \varphi \; \theta \; \psi \;\; \dot{r}_1 \; \dot{r}_2 \cdots \dot{r}_{N_{\mathrm{mod}es}} \;\; r_1 \; r_2 \cdots r_{N_{\mathrm{mod}es}} \;\; q_1 \; q_2 \cdots q_{N_{\mathrm{mod}es}} \right]^T \quad (49)$$

and **E** represents the 'expression vector' of an expression E, then,

$$E = \mathbf{E}^T \mathbf{v} \quad (50)$$

It may be noted that if $E_1 = \mathbf{E}_1{}^T \mathbf{v}$ and $E_2 = \mathbf{E}_2{}^T \mathbf{v}$, then $aE_1 + bE_2 = \left(a\mathbf{E}_1 + b\mathbf{E}_2 \right)^T \mathbf{v}$.

Since we note that in (10) and (20), and in fact everywhere because of linearization of the equations, the expressions only get summed/subtracted. Hence we can easily define and deal with the expression vectors and not the expressions as a wholeand keep the terms of the state variables separate.

Hence we determine the expression vectors for the forces and moments due to thrust, engine inertia, aerodynamic forces and gravity and use then in (10) and (20) to get equations of the form $\mathbf{E}_{eqn} = 0$, where $\mathbf{E}_{eqn}$ is the expression vector corresponding to the expression on one side of the equation when all the terns are taken to one side. Now the task reduces to adding new rows to matrices **A** and **B**, and new elements to $\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$ and **D** using the elements of $\mathbf{E}_{eqn}$.

Flexible and easily understandable code: The code consists of separate .m files for each objects described above. For example, The main time loop calls separate functions to evaluate the contributions of each of the forcing factors to a global variable for describing the net distribution of forces and moments on the missile body; separate functions for including elements into the matrices of state-space equation (40) contributed by the rigid body equations (10) and the flexibility of the beam (20). This enables us to turn on or off some particular forcing according to our will.

The MATLAB code has been provided in Appendix-I

Schematic flowchart for simulation of the system

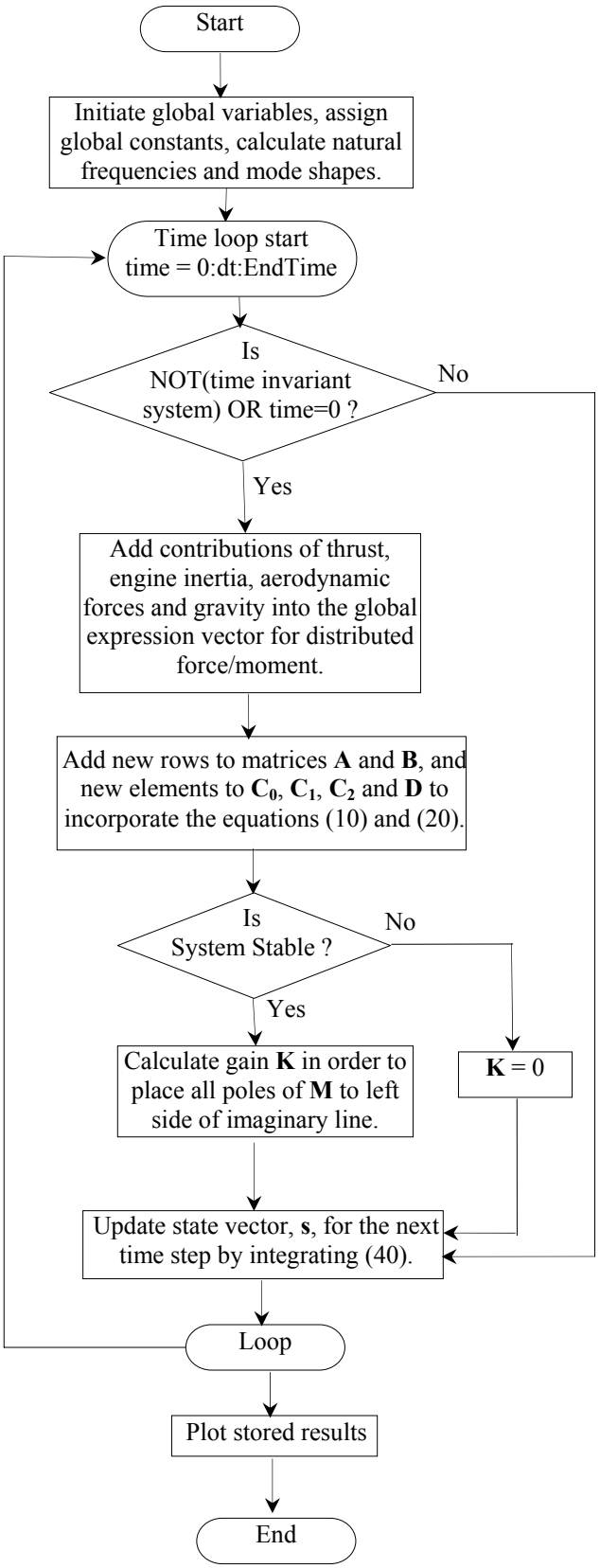


fig - 7

3.2 Results

The system was simulated for several cases and plots were made to demonstrate the results. The following sections describe the results:

3.2.1 Simulation – 1

The simulation was performed with the following parameters and specifications:

$L_0 = 1$; $L_C = 0.5$; $L_R = 0.04$; $L_A = 0.5$; $A = \pi\, 0.1^2 / 4$;
$\rho = 2600$; $E = 10^7$;
$m_R = 0.1$; $m_M = \rho\, L_0\, A$;
ψ_0, θ_0 and φ_0
$U_0 = 1000$; $T_C = T_S = 10^7$;
The system matrices are time invariant;
$N_{modes} = 10$; $N_{slosh} = 0$; $N_S = 29$;

The poles of the open loop system, i.e. eigenvalues of $\mathbf{A}^{-1}\mathbf{B}$ are found to be,

```
1.0e+005 *

      0
      0
      0
      0
      0
-0.0284 + 2.7800i
-0.0284 - 2.7800i
-0.0233 + 2.2731i
-0.0233 - 2.2731i
-0.0188 + 1.8188i
-0.0188 - 1.8188i
-0.0148 + 1.4154i
-0.0148 - 1.4154i
-0.0114 + 1.0625i
-0.0114 - 1.0625i
-0.0084 + 0.7596i
-0.0084 - 0.7596i
-0.0059 + 0.5066i
-0.0059 - 0.5066i
-0.0039 + 0.3033i
-0.0039 - 0.3033i
-0.0026
-0.0013 + 0.0333i
-0.0013 - 0.0333i
-0.0024 + 0.1487i
-0.0024 - 0.1487i
      0
      0
      0
```

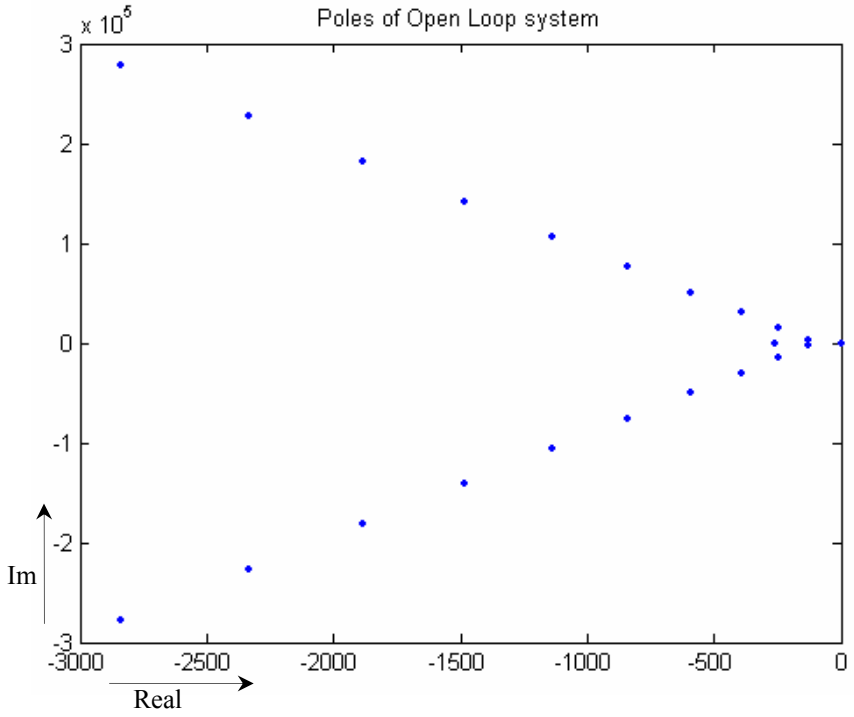


fig - 8

Hence, the system is stable. Thus the gain **K** is taken to be zero.

The natural frequencies and the corresponding *A* and *B* (of equation (13)) are found to be:

```
Mode 1: beta=4.7300407449, A=-5.764552e+001, B=5.663685e+001, omg=5.804464e+003
Mode 2: beta=7.8532046241, A=-1.285985e+003, B=1.286984e+003, omg=1.600023e+004
Mode 3: beta=10.9956078380, A=-2.980687e+004, B=2.980587e+004, omg=3.136684e+004
Mode 4: beta=14.1371669412, A=-6.897044e+005, B=6.897054e+005, omg=5.185100e+004
Mode 5: beta=17.2787595947, A=-1.596026e+007, B=1.596026e+007, omg=7.745643e+004
Mode 6: beta=20.4203522483, A=-3.693315e+008, B=3.693315e+008, omg=1.081829e+005
Mode 7: beta=23.5619449019, A=-8.546586e+009, B=8.546586e+009, omg=1.440306e+005
Mode 8: beta=26.7035375555, A=-1.977739e+011, B=1.977739e+011, omg=1.849992e+005
Mode 9: beta=29.8451302091, A=-4.576625e+012, B=4.576625e+012, omg=2.310890e+005
Mode 10: beta=32.9867228627, A=-1.059063e+014, B=1.059063e+014, omg=2.822999e+005
```

The initial condition was taken to be that the missile is disturbed in its first mode and released. Hence all the state variables, except q_1 is taken to be zero. And $q_1 = 0.001$ was taken initially. The simulation was performed from 0 to 0.01s, with $dt = 10^{-7}$.

The shape of the missile at 10 different instants of time were captured:

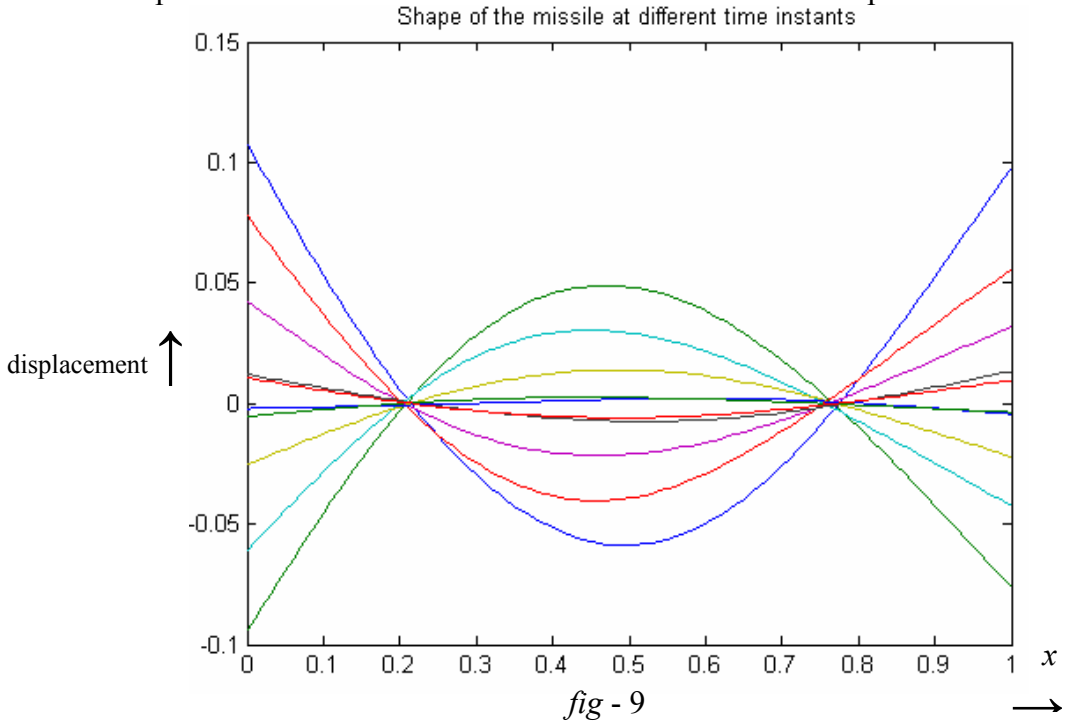


fig - 9

The displacement at 3 points on the missile with time:

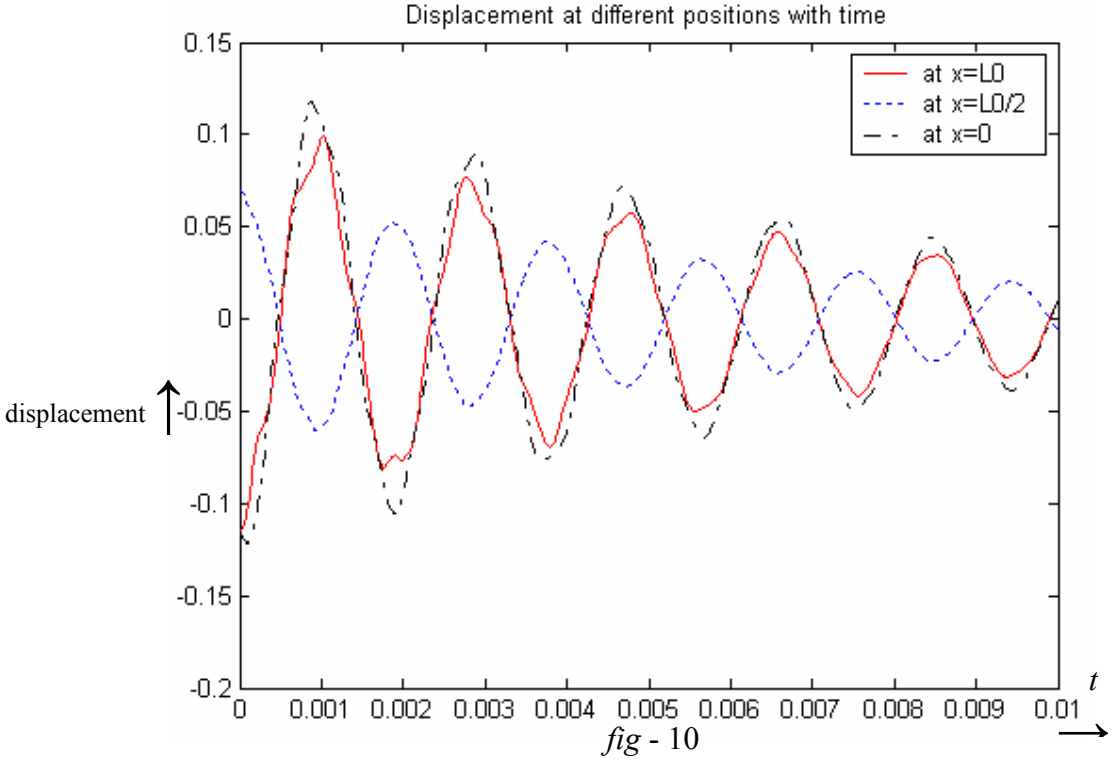


fig - 10

It may be observed that the missile primarily tries to deform in the first mode, while the other modes get excited slightly. The vibration amplitude dies down with time due to presence of viscous damping.

3.2.2 Simulation - 2

The same missile with the same specifications is again used, except that U_0 is changed to $U_0 = 5000$.
Hence the modes and natural frequencies remain the same. But the poles of open loop system change. The poles of the open loop system, i.e. eigenvalues of $\mathbf{A}^{-1}\mathbf{B}$ are now found to be,

```
1.0e+005 *

       0
       0
       0
       0
       0
-0.0319 + 2.7801i
-0.0319 - 2.7801i
-0.0267 + 2.2732i
-0.0267 - 2.2732i
-0.0223 + 1.8189i
-0.0223 - 1.8189i
-0.0183 + 1.4155i
-0.0183 - 1.4155i
-0.0149 + 1.0625i
-0.0149 - 1.0625i
-0.0119 + 0.7597i
-0.0119 - 0.7597i
-0.0094 + 0.5068i
-0.0094 - 0.5068i
-0.0075 + 0.3034i
-0.0075 - 0.3034i
-0.0051
-0.0080 + 0.0283i
-0.0080 - 0.0283i
-0.0059 + 0.1495i
-0.0059 - 0.1495i
       0
       0
       0
```

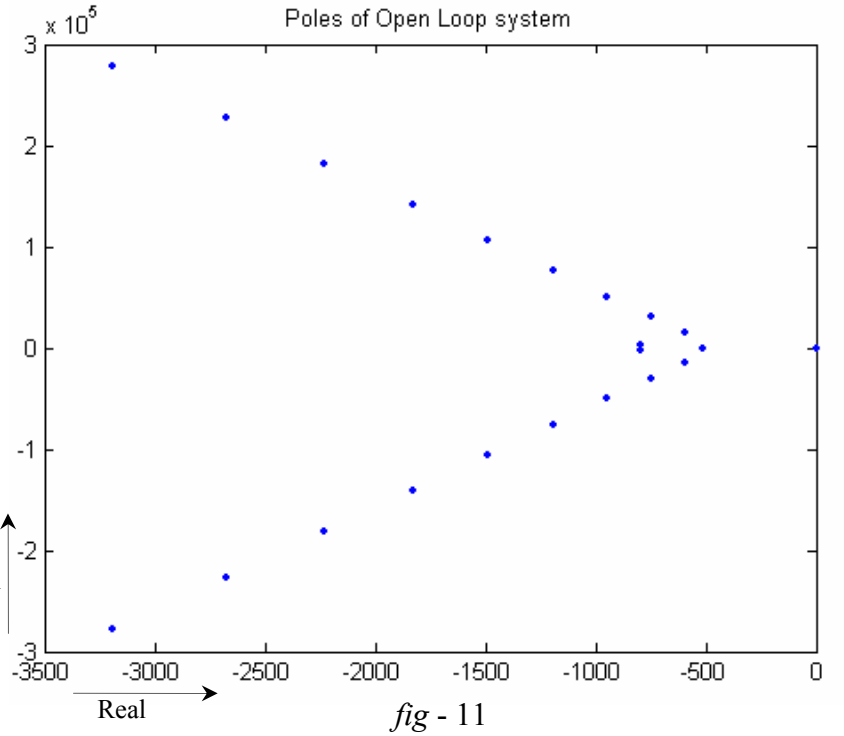


fig - 11

The system now it is excited initially in the 3rd and 5th modes by amounts $q_3 = 5\text{x}10^{-9}$ and $q_5 = 10^{-9}$ respectively.

The simulation was performed from 0 to 0.0001s, with $dt = 10^{-8}$.

The shape of the missile at 10 different instants of time:

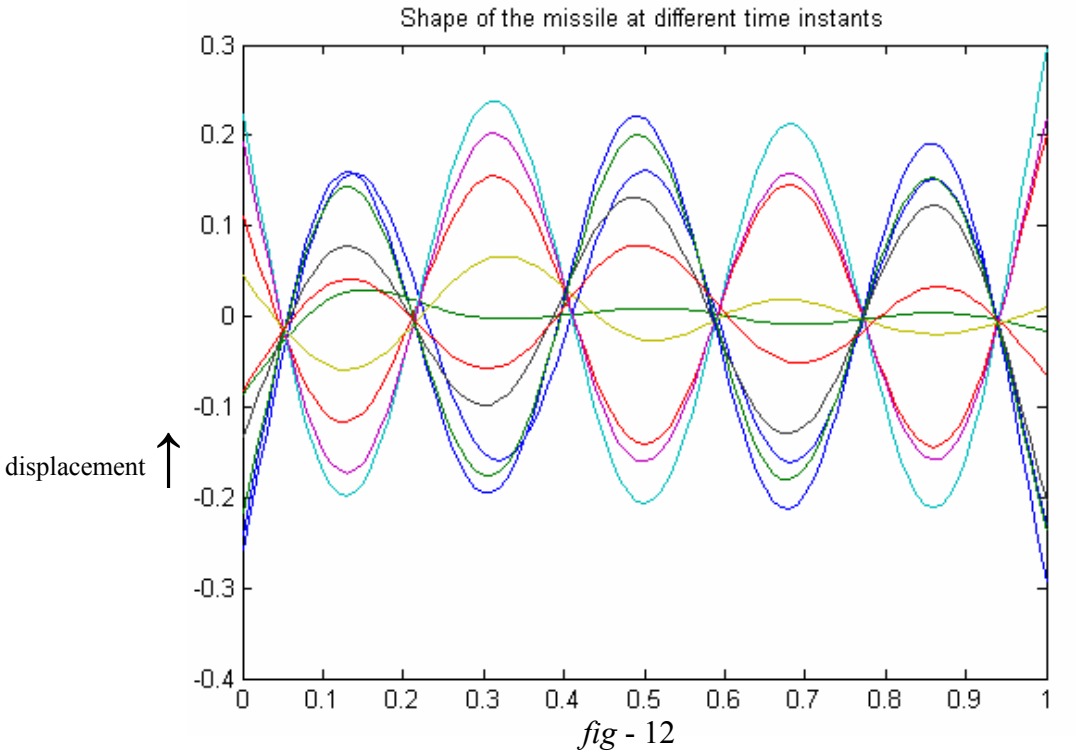


fig - 12

The displacement at 3 points on the missile with time:

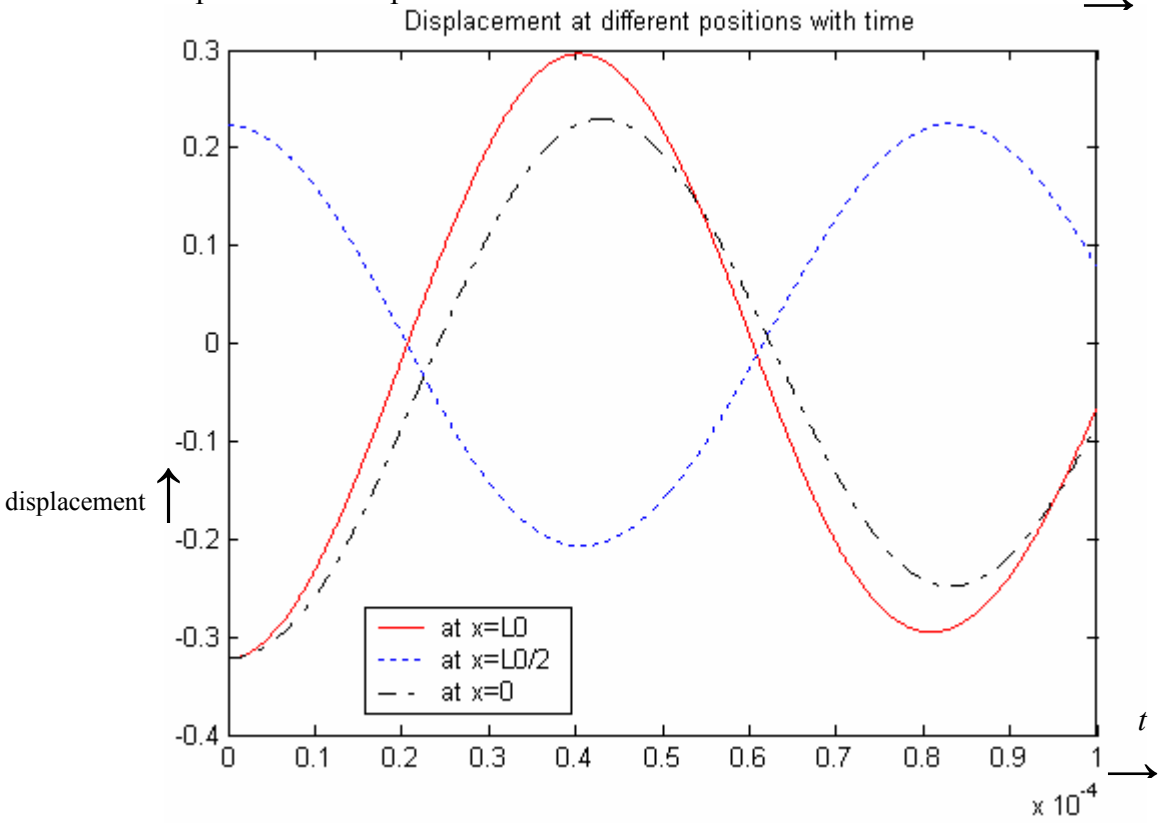


fig - 13

3.2.3 Instability at larger U_0

The system was again set with parameters same as before, except that now U_0 = 10000.
Now the poles of the open loop system was found to be,

```
1.0e+005 *

       0
       0
       0
       0
       0
-0.0363 + 2.7803i
-0.0363 - 2.7803i
-0.0311 + 2.2734i
-0.0311 - 2.2734i
-0.0267 + 1.8191i
-0.0267 - 1.8191i
-0.0226 + 1.4157i
-0.0226 - 1.4157i
-0.0192 + 1.0628i
-0.0192 - 1.0628i
-0.0163 + 0.7598i
-0.0163 - 0.7598i
-0.0138 + 0.5071i
-0.0138 - 0.5071i
-0.0121 + 0.3032i
-0.0121 - 0.3032i
 0.0244
-0.0332 + 0.0404i
-0.0332 - 0.0404i
-0.0095 + 0.1528i
-0.0095 - 0.1528i
       0
       0
       0
```

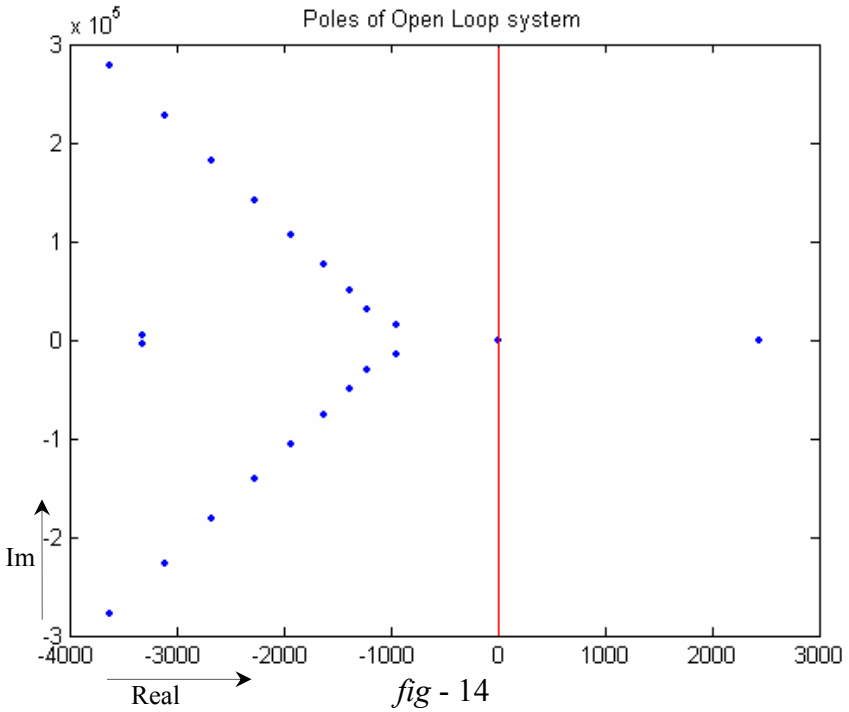


fig - 14

Here we observe that one of the poles (0.0244) lies no the right side of the imaginary axis. Hence the system is unstable and the solution diverged on integrating.

3.2.4 Root locus with variation of U_0

Keeping everything else same as mentioned in Simulation-1, we varied U_0 from 100 to 20000 in order to observe how the poles of the open loop system vary with it.

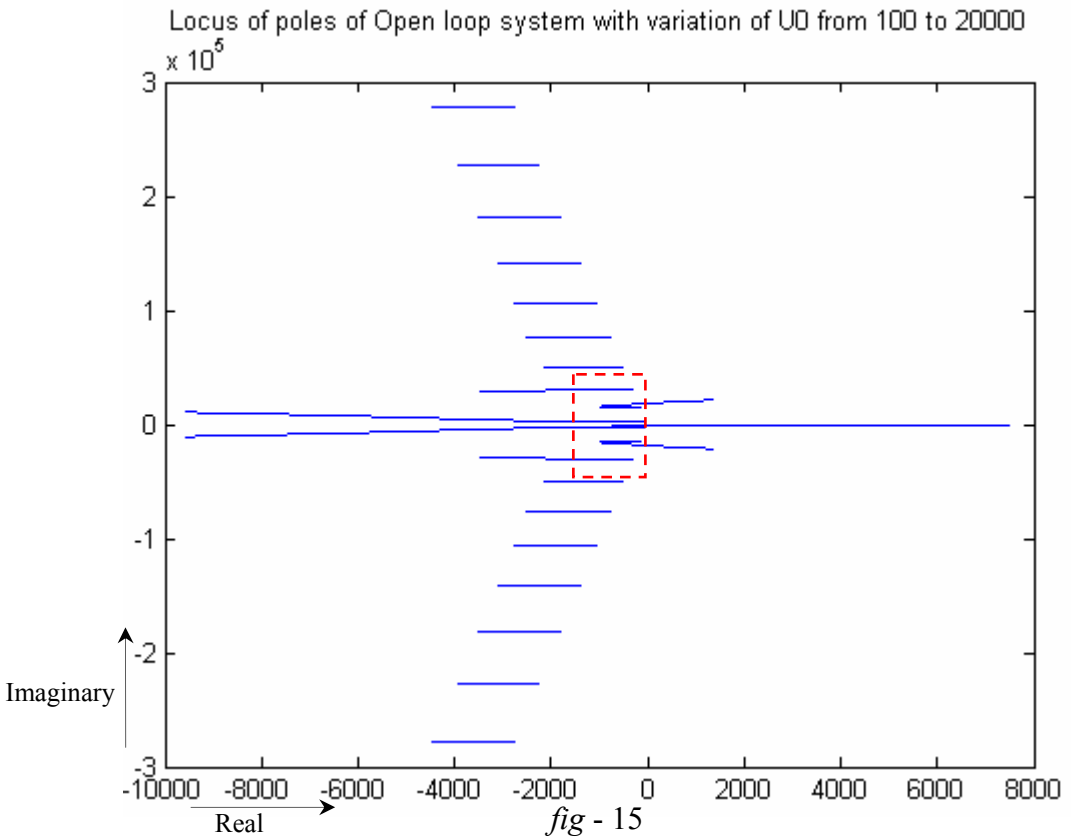


fig - 15

Magnified view of the above root locus at the region marked by the dotted box:

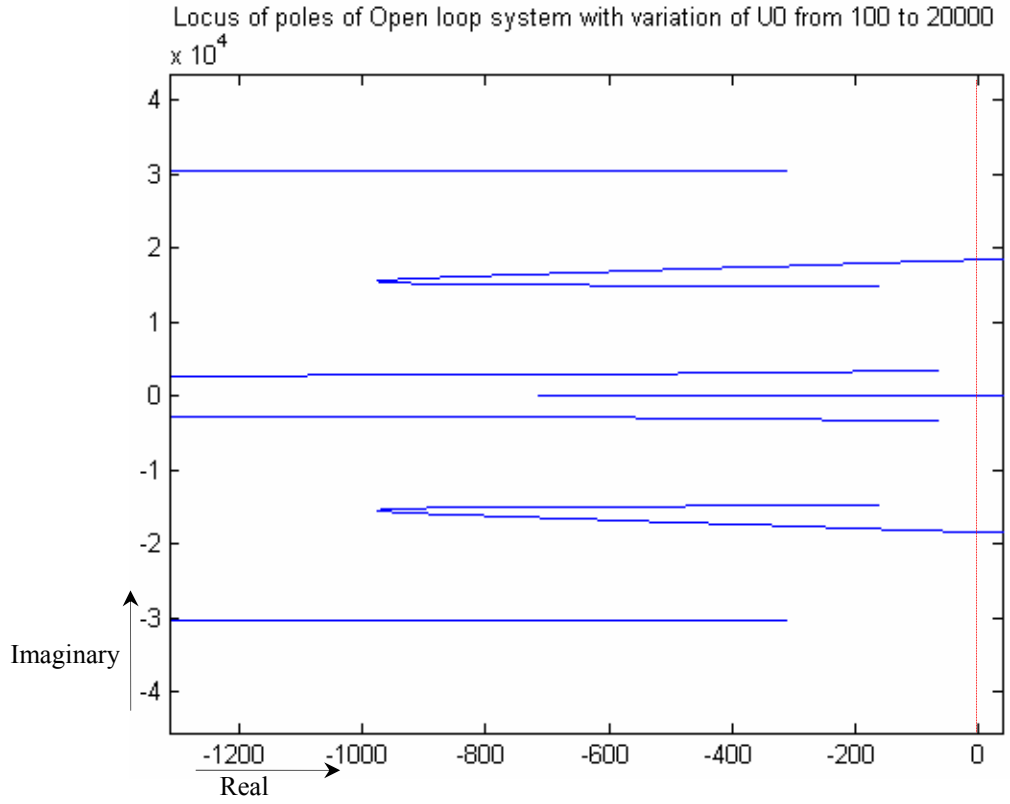


fig - 16

3.2.5 Root locus with variation of T_C

Keeping everything else same as mentioned in Simulation-1, we varied T_C from in order to observe how the poles of the open loop system vary with it.

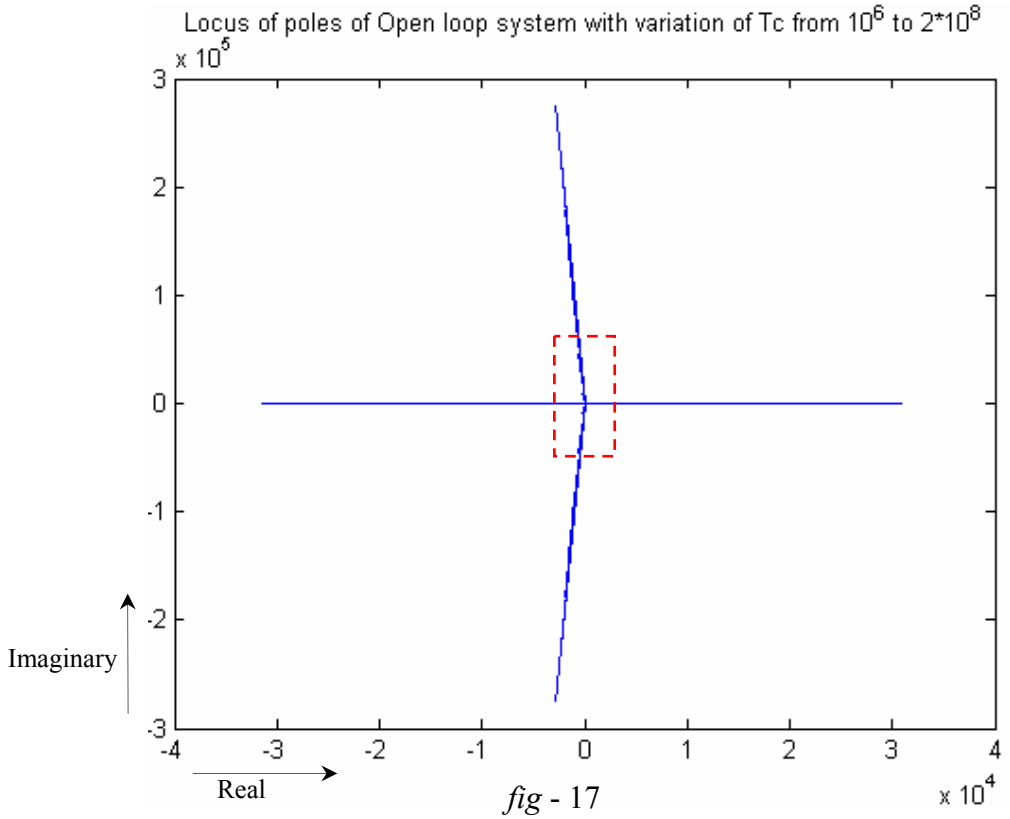


fig - 17

The plot is magnified at the region marked by the dotted box:

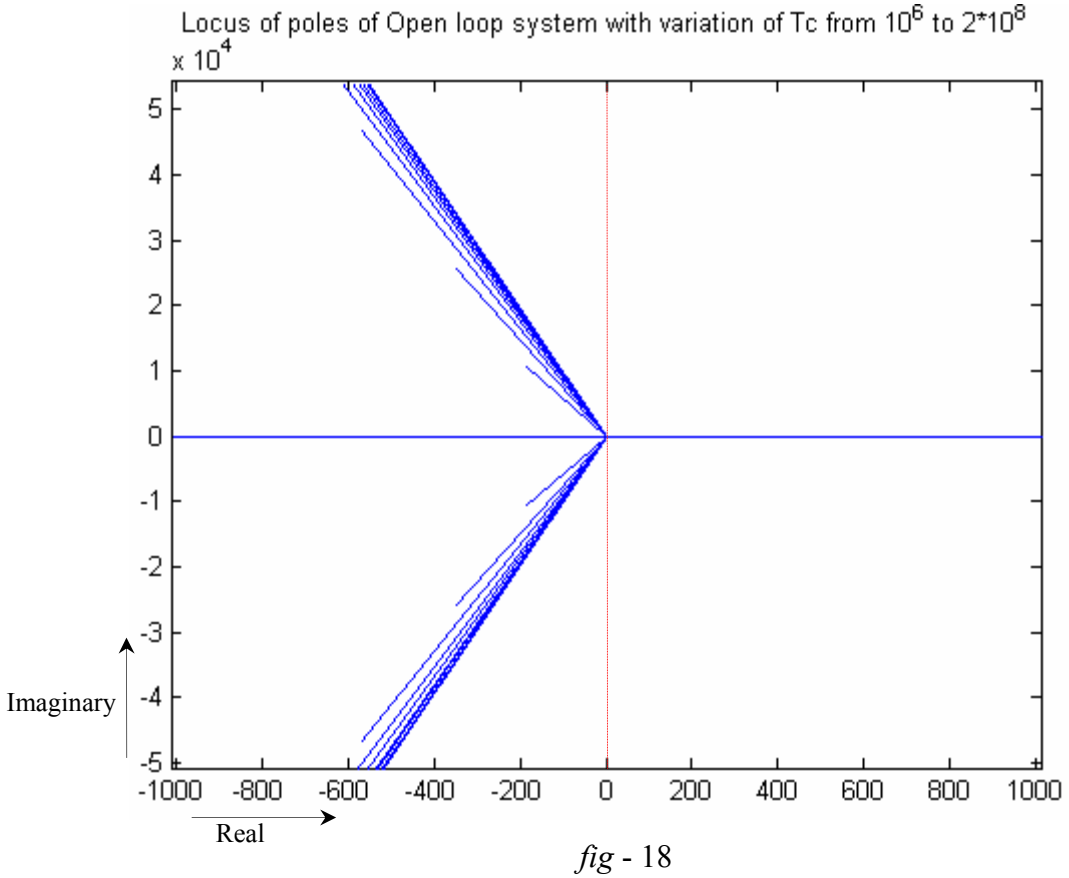


fig - 18

3.2.6 Attempt of finding a suitable gain, K, for stabilizing the system in an unstable situation

The system with $U_0 = 1000$; $T_C = T_S = 10^7$ was found to be unstable. In order to stabilize this system we tried to implement an integral feedback control loop.
As mentioned earlier, for stabilizing the closed loop system we need to place the poles of the matrix $\boldsymbol{M}$ in eqn (45) to the left of the imaginary axis.
Attempts to find a suitable **K** by Newton-Raphson iteration or Genetic Algorithm failed. However a last method of searching for individual elements of **K** and assigning score according to ability of the element to push poles to the left succeeded.
The poles of **M** with zero **K** were,

```
  1.0e+005 *

        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
        0
  -0.0363 + 2.7803i
  -0.0363 - 2.7803i
  -0.0311 + 2.2734i
  -0.0311 - 2.2734i
  -0.0267 + 1.8191i
  -0.0267 - 1.8191i
  -0.0226 + 1.4157i
  -0.0226 - 1.4157i
  -0.0192 + 1.0628i
  -0.0192 - 1.0628i
  -0.0163 + 0.7598i
  -0.0163 - 0.7598i
  -0.0138 + 0.5071i
  -0.0138 - 0.5071i
  -0.0121 + 0.3032i
  -0.0121 - 0.3032i
  -0.0095 + 0.1528i
  -0.0095 - 0.1528i
   0.0244
  -0.0332 + 0.0404i
  -0.0332 - 0.0404i
        0
        0
        0
        0
        0
        0
```

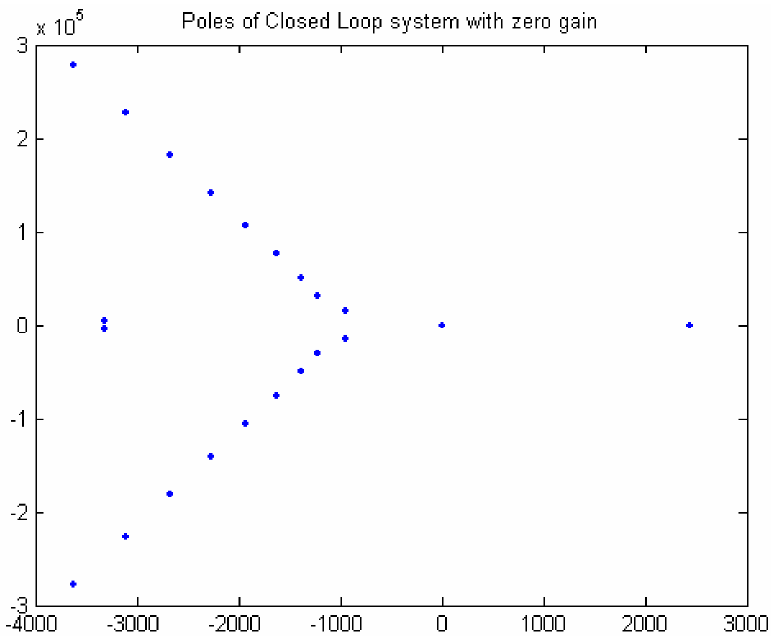


fig - 19

The pole to the right of the imaginary line is the cause of instability.

The **K** determined by the foresaid method was,

K = [0 9.6006e+024 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]^{T}

Note that one of the elements of **K** is very large, while the others are *almost* zero.

This **K** shifted the new poles of **M** as shown:

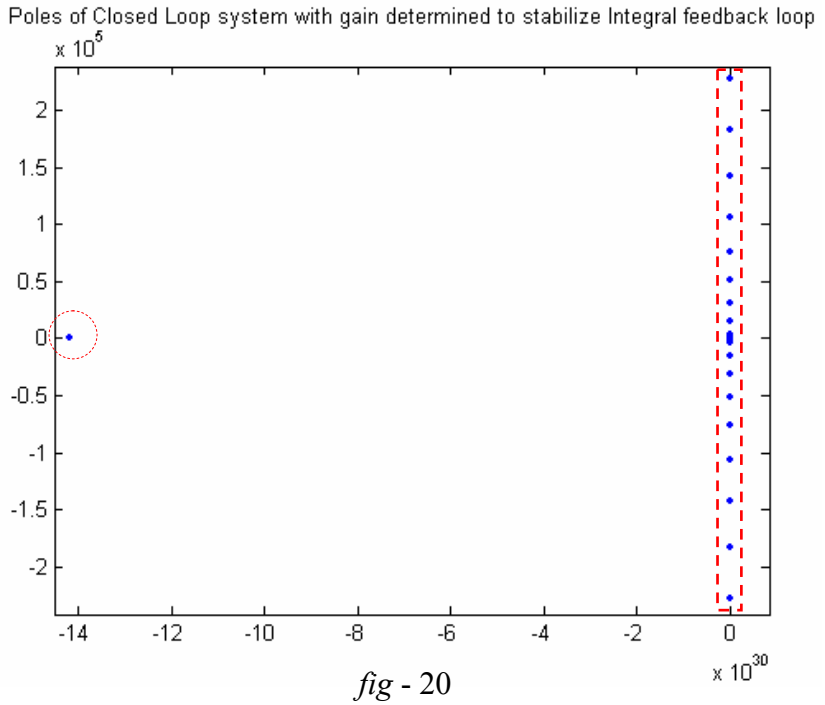


fig - 20

It may be noticed that one pole has been shifted to a location of the order of about 10^{31} to the left (marked by dotted circle). However the other poles remain at normal location, as can be seen on magnifying the zone marked by the dotted rectangle:

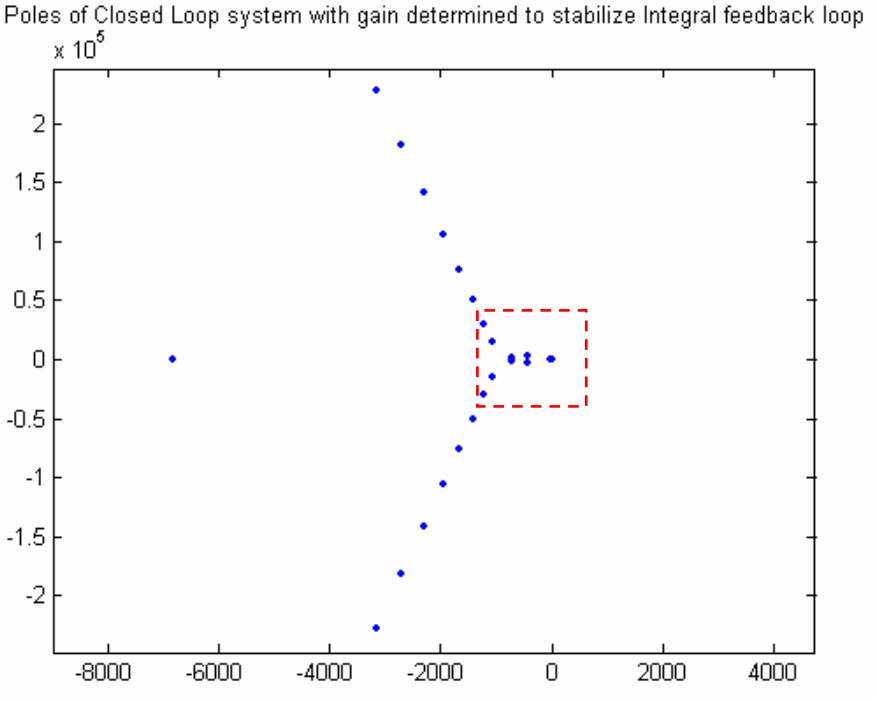


fig - 21

On further magnification:

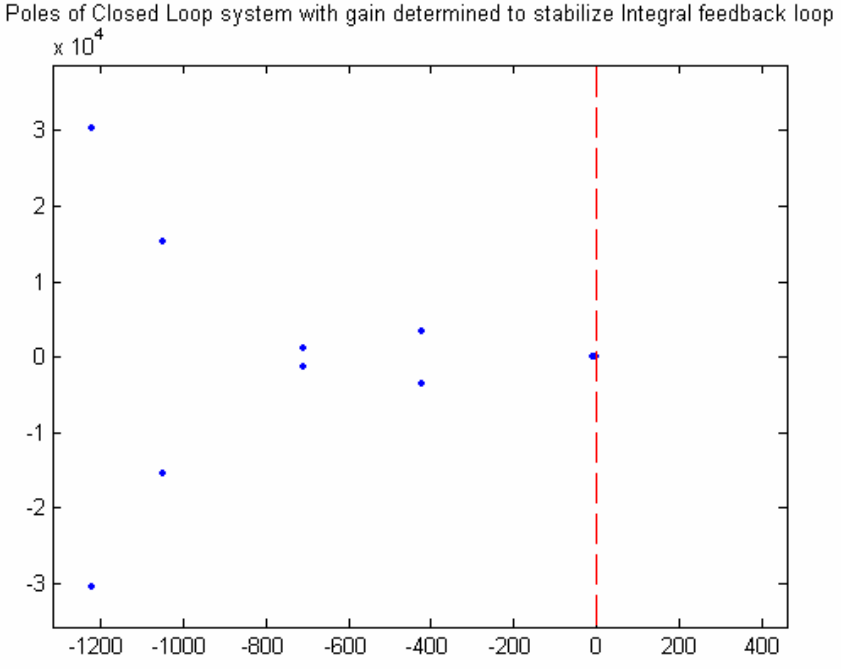


fig - 22

Hence we can see that no poles are any more to the right of the imaginary line.

However because of the presence of such a high gain the First Order Runge kotta method for integrating (40) failed to converge.
Convergence was however observed on using very small time steps. But then the net tome over which the integration can be performed reduced markedly. Hence the values of the state variables almost remained un-changed.

3.3 Discussions and Conclusions

1. The dynamics of short-period motion of missile was implemented in MATLAB code taking into account the flexibility of the missile and forcing due to factors like thrust, gravity, engine inertia, and aerodynamics.
2. The code was implemented keeping in mind it's easy changeability and flexibility.
3. The model was developed considering the forces and their perturbation components to be acting only in the pitch plane.
4. Though a control scheme could be developed and a gain for stabilizing an unstable case could also be developed, the gain being of extreme high magnitude resulted in divergence of the simulation.

4 Study on stability of flow

4.1 Origin of Turbulence and ways to analyze stability of a flow:

Flow induced vibration is caused in structures by forcing due to time variant pressure acting on the surface of the structure. If the flow is imagined to be a linear superposition of a steady state laminar flow and a perturbation flow field, the laminar flow won't cause the forcing on the structure as it acts as a time invariant pressure on the structure's surface. It is the perturbation flow field that varies with time and hence produces forcing on the structure's surface.
This same perturbation flow field is the sole cause of Turbulence in the flow. The origin of the perturbation flow field may be something like a very small disturbance caused in the laminar flow field. A flow is said to be stable if for any small initial disturbance (i.e. perturbation) added to the laminar flow field, the perturbation flow field gradually dies down with time. It will be termed as a Transitional flow if the perturbation flow field gets magnified with time, which in turn gives rise to Turbulence. Hence our primary approach will be to investigate the nature of variation of a perturbation flow field with time.

4.2 The Orr-Somerfeld equation for two-dimensional flow

A two-dimensional incompressible flow is governed by the Navier Stoke's equations,

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}=\frac{1}{\rho}F_X-\frac{1}{\rho}\frac{\partial p}{\partial x}+\nu\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right) \tag{51a}$$

$$\frac{\partial v}{\partial t}+u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}=\frac{1}{\rho}F_Y-\frac{1}{\rho}\frac{\partial p}{\partial y}+\nu\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right) \tag{51b}$$

and the continuity equation,

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0 \tag{52}$$

Let us consider a steady-state laminar flow over an infinitely long plate.

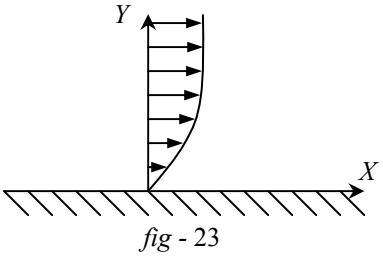


fig - 23

For this flow,

$$\left.\begin{aligned} u &= U(y) \\ v &= 0 \\ p &= P(x) \end{aligned}\right\} \tag{53}$$

Now let the perturbation field be described by the perturbation components denoted by a 'prime' upon the steady-state laminar variables.
Hence the final flow field will be described by,

$$\left.\begin{aligned} u &= U(y) + u'(x, y, t) \\ v &= v'(x, y, t) \\ p &= P(x) + p'(x, y, t) \end{aligned}\right\} \tag{54}$$

Now, both the steady-state laminar field (53) and the superposed field (54) satisfies equation (51) and (52). Hence by substituting them in (51) and (52) and performing some simplification, we obtain the equations governing the perturbation flow field,

$$\frac{\partial u'}{\partial t} + U\frac{\partial u'}{\partial x} + v'\frac{\partial U}{\partial y} + \frac{1}{\rho}\frac{\partial p'}{\partial x} = \nu \nabla^2 u' \tag{55a}$$

$$\frac{\partial v'}{\partial t} + U\frac{\partial v'}{\partial x} + \frac{1}{\rho}\frac{\partial p'}{\partial y} = \nu \nabla^2 v' \tag{55b}$$

$$\frac{\partial u'}{\partial x} + \frac{\partial v'}{\partial y} = 0 \tag{55c}$$

From these equations, with a known velocity profile *U(y)*, we may obtain solutions for *u′*, *v′*, and *p′*. Our present aim will be to determine for a given velocity profile *U(y)*, the coefficient of viscosity *ν* and the boundary & initial conditions of the perturbation fields, whether or not the perturbation components die down with time.
In order to satisfy eqn. (55c), we define a stream function $\psi(x, y, t)$ such that

$$u' = \frac{\partial \psi}{\partial y} \text{ and } v' = -\frac{\partial \psi}{\partial x} \tag{56}$$

It is now assumed that the disturbances, and hence ψ is superposition of several periodic disturbances (periodic in *x*) propagating along the direction of flow.
Hence, we substitute,

$$\psi(x, y, t) = \phi(y)\, e^{i(\alpha x - \beta t)} \tag{57}$$

ψ is said to be periodic on *x* with frequency α and wavelength $\lambda_X = \frac{2\pi}{\alpha}$. Though α can be assumed to be real, β being the time frequency should be assumed to be complex in order to keep the possibility of non-periodic magnification or decay of ψ with time. The final ψ will be superposition of all the solutions of ψ.
We define the complex velocity of propagation of the disturbance as

$$c = \frac{\beta}{\alpha} = c_r + i\, c_i \tag{58}$$

$\therefore \psi = \phi(y)\, e^{\alpha[c_i t + i(x - c_r t)]}$
It may be noted that for $c_i < 0$, the solution of ψ dies down with time, and hence so does the perturbation components. Thus the flow is stable for $c_i < 0$ and tends to become turbulent for $c_i > 0$.

We put *u′* and *v′* in terms of ϕ, α and β in (55a) and (55b) and eliminate *p′* to obtain a single equation. We non-dimensionalize the equations by redefining *y* as $\frac{y}{\delta}$, *U* as $\frac{U}{U_m}$, *c* as $\frac{c}{U_m}$, where U_m is the free-stream velocity of the flow and δ is the boundary layer thickness.

The non-dimensionalized equation hence obtained is called the Orr-Sommerfeld equation:

$$(U-c)(\phi''-\alpha^2\phi)-U''\phi=-\frac{i}{\alpha R}\left(\phi''''-2\alpha^2\phi''+\alpha^4\phi\right) \tag{59}$$

where, $R=\dfrac{U_m\delta}{\nu}$ is the Reynolds Number.

A trivial solution to eqn.(59) is $\phi=0$. For non-trivial solution, for a given α and *R*, we obtain an eigenvalue of *c* and the corresponding eigenfunction ϕ. Hence our immediate target is to find an eigenvalue solution of (59).

4.3 Boundary conditions for the Orr-Sommerfeld equation

4.3.1 Boundary conditions for flow over a rigid, static surface

For this case,

At $y = 0$,

$v' = 0$ and $u' = 0$

$\Rightarrow \quad \phi(0)=0$ and $\phi'(0)=0$

At $y\to\infty$,

$v' = 0$ and $u' = 0$

$\Rightarrow \quad \phi(\infty)=0$ and $\phi'(\infty)=0$

4.3.2 Boundary conditions for two-dimensional flow over a beam

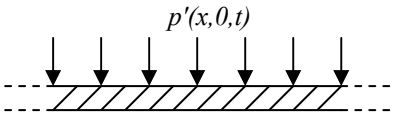


fig - 24

We consider flow over an infinite flexible beam. The net unbalanced force acting on the beam is *p'*. Hence, the differential equation governing the motion of the beam is given by,

$$\lambda\frac{\partial^2 w}{\partial t^2}=-p'(x,0,t)-EI\frac{\partial^4 w}{\partial x^4} \tag{60}$$

Where, *w* denotes the displacement of the beam in *Y* direction, λ is mass per unit length of the beam. The forcing on the beam is due to the time-variant perturbation pressure. It is to be noted in this case that equations (55), (59) and (60) gets coupled. One way of solving the equation will be as follows. As *p'* is of the form of $e^{i(\alpha x-\beta t)}$, we may write for eqn. (60),

$$w(x,t)=k\,e^{i(\alpha x-\beta t)} \tag{61}$$

Substituting this *w* in (60) and hence substituting the hence obtained *p'(x,0,t)* into (55a), and Substituting *v'* in terms of ϕ in the same (55a) equation, all at $y = 0$, we obtain,

$$k=\frac{\rho\left[\,i\,\alpha\,U'(0)\,\phi(0)-(\alpha^2\nu-i\,\beta)\,\phi'(0)+\nu\phi''(0)\right]}{i\,\alpha\left(\lambda\beta^2-EI\alpha^4\right)} \tag{62}$$

Hence the boundary conditions become,

At y = 0,

$v' = \dot{w}$ and $u' = 0$

$\Rightarrow \quad \phi(0) = -\frac{\beta}{\alpha}\bar{k}$ and $\phi'(0) = 0$, where $\bar{k}$ is the non-dimensionalized *k*.

At y→∞,

$v' = 0$ and $u' = 0$

$\Rightarrow \quad \phi(\infty) = 0$ and $\phi'(\infty) = 0$

Hence, as it can be seen, one of the boundary conditions is a bit more complex involving ϕ, ϕ', ϕ'' and *c*.

The numerical technique for solving the Orr-Sommerfeld equation for the eigenvalues and eigenfunctions in either of the cases will remain similar, except that the boundary conditions are modified. However till date we have only investigated the case 'A', i.e. the case of flow over a rigid, static plate.

4.4 Numerical methods attempted for solving the Orr-Sommerfeld equation

4.4.1 Galerkin's Method

This method, though can handle the case of flow over a rigid, static plate satisfactorily, its application in solving the case of flexible plate is difficult. The method for the case of flow over a rigid, static plate is described below in brief.

We denote eqn.(59) by $\Gamma(\phi) = 0$, where Γ denotes the operator. We write ϕ as a linear superposition of several functions ϕ_i that satisfy the boundary conditions given in 4.3.1 Such functions were chosen to be of the form, $\phi_i = y^{\eta} e^{-\mu y}$.

Hence we write,

$$\phi(y) = \sum_{i=1}^{n} \xi_i \phi_i(y)$$

We now define an error,

$$e(y) = \Gamma\left(\sum_{i=1}^{n} \xi_i \phi_i(y) \right) \tag{63}$$

We need to choose the values of ξ_i in such a way that this error is minimized. We do that by solving the set of *n* equations,

$$\langle e, \phi_i \rangle = 0, \; i = 1 \text{ to } n \tag{64}$$

where, $\langle \chi_i, \chi_j \rangle$ denotes the inner product given by,

$$\langle e, \phi_i \rangle = \int_0^{\infty} e \, \phi_i \, dy$$

The n equations in (64) have ξ_i as the unknowns and can be represented in the matrix form as,

$$[M]\begin{bmatrix}\xi_1\\ \vdots\\ \xi_n\end{bmatrix}=0 \tag{65}$$

Where the matrix [M] contains the unknown c.
For non-trivial solution of this equation, we must have,

$$\left|[M]\right|=0 \tag{66}$$

In general we'll obtain n solutions for c. We should chose the one for which the ξ_i s are such that $\Gamma\left(\sum_{i=1}^{n}\xi_i\,\phi_i(y)\right)$ is minimum.

This method was implemented in Mathematia 5.1 and a rather unsatisfactory result was obtained probably due to the following reasons:

i. Due to limitation of computational power, n had to be limited to 4 in order to get a satisfactory and accurate integration value of the inner product and solution of c from (66).
ii. The choice of η and μ in choosing the functional forms were done arbitrarily.

The Mathematica code has been given in Appendix-II.

We determined the eigenvalues c for different α and R and plotted the contour in the α-R plane for which $c_i = 0$. This contour will mark the margin between the stable and unstable zones of α and R. However the contour $c_i = 0$ could not be found satisfactorily in the fist quadrant of α-R plane. But on plotting a contour plot of c_i , the following was obtained. The plot demonstrates that the basic shape of the standard results is being approached by the solution:

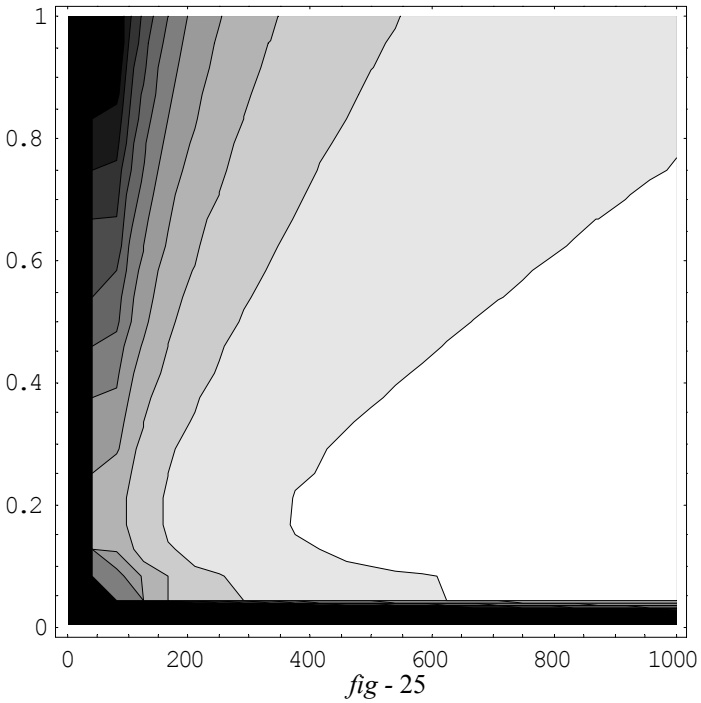


fig - 25

The horizontal axis denotes R and the vertical axis denotes α.

4.4.2 'Automated search of eigenvalues' – Integration by Runge-Kutta

This method primarily proposed by Betchov & Criminale [7], deals the regions above and below the boundary layer separately. We first investigate the Orr-Sommerfeld equation (59) for $y > 1$. In this region, the non-dimensionalised velocity is $U = 1$. Hence the equation is modified as,

$$(1-c)(\phi'' - \alpha^2\phi) = -\frac{i}{\alpha R}\left(\phi'''' - 2\alpha^2\phi'' + \alpha^4\phi\right) \text{ for } y > 1 \tag{67}$$

This equation being 4th order linear in ϕ with constant coefficient has simple analytical solution given by,

$$\phi(y) = \sum_{j=1}^{4} A_j e^{p_j y} \tag{68}$$

where, p_j are the roots of,

$$(1-c)\left(p^2 - \alpha^2\right) = -\frac{i}{\alpha R}\left(p^2 - \alpha^2\right)^2 \tag{69}$$

Hence, $p_1 = \alpha$, $\quad p_1 = -\alpha$

$$p_3 = \alpha\left[1 + i\frac{R}{\alpha}(1-c)\right]^{1/2}, \qquad p_3 = -\alpha\left[1 + i\frac{R}{\alpha}(1-c)\right]^{1/2}$$

From boundary condition at ∞, as y→∞, $\phi = 0$ and $\phi' = 0$.
Hence we have, $A_1 = A_3 = 0$ for $y > 1$.
$\therefore \phi(y) = A_2 e^{p_2 y} + A_4 e^{p_4 y}$.

Now we argue that, since for $y > 1$, the solution is a linear superposition of two modes $e^{p_2 y}$ and $e^{p_4 y}$, the solution for $y < 1$ will also be linear superposition of these two same modes. Hence now our task is to find the solution of these two modes in the region $y < 1$. We attain this by performing two integration passes from $y = 1$ to $y = 0$ independently. We used Runge-Kutta method for this numerical integration.

Integration Pass – I:
We start from $y = 1$ with $A_2 = 0$ and $A_4 = 1$(or some other value). Therefore we take the initial values $\phi(1) = e^{p_4}$, $\phi'(1) = p_4 e^{p_4}$, $\phi''(1) = p_4^2 e^{p_4}$ & $\phi'''(1) = p_4^3 e^{p_4}$ and move on integrating towards $y = 0$. Let the solution obtained in this process be called $\phi_I(y)$.

Integration Pass – II:
Similarly with $A_2 = 1$(or some other value) and $A_4 = 0$ we obtain the second pass integration $\phi_{II}(y)$.

Hence the final solution is of the form $\phi(y) = a_I \phi_I(y) + a_{II}\phi_{II}(y)$.

From the boundary conditions at $y = 0$,

$$\phi(0) = a_I\phi_I(0) + a_{II}\phi_{II}(0) = 0 \quad \text{and} \quad \phi'(0) = a_I\phi_I'(0) + a_{II}\phi_{II}'(0) = 0,$$

for non-trivial solution of a_I and a_{II}, we must have,

$$\begin{vmatrix} \phi_I(0) & \phi_{II}(0) \\ \phi_I'(0) & \phi_{II}'(0) \end{vmatrix} = 0 \tag{70}$$

It is to be noted that the only unknown in (70) for a given α and R is c. Hence from 20 we obtain the eigenvalue c. The corresponding eigenvector gives a_I and a_{II}, and hence the final solution of $\phi(y)$.

Search for Eigenvalue in c-plane:

However it was not possible to solve c explicitly from (70). Hence we assumed some c and determined the solutions from the two integration passes, $\phi_I(y)$ and $\phi_{II}(y)$.
For those particular solutions we defined,

$$f(c) = \begin{vmatrix} \phi_I(0) & \phi_{II}(0) \\ \phi_I'(0) & \phi_{II}'(0) \end{vmatrix}$$

As we know *f(c)* should converge to 0, we used an iteration scheme [7] as follows.
We start with an arbitrary value of c and some small value of δc and go on updating it using the following iteration,

$$\Delta c = -\lambda \left(\frac{f(c+\delta c)}{f(c)} - 1 \right)^{-1} \delta c$$
$$c \leftarrow c + \Delta c$$
$$\delta c \leftarrow \mu \Delta c$$

where, λ = 1, 0.8 or 0.5 depending on whether the convergence is fast, moderate or slow, and, $\mu = \frac{1}{4}$. The iteration continues till Δc reaches a substantially small value.

The procedure was implemented in MATLAB and the code has been provided in Appendix-III.

Like before, we once again searched for the contour of $c_i = 0$ in the α-R plane. The results obtained, though not satisfactory, is described in the following plot of 20 points:

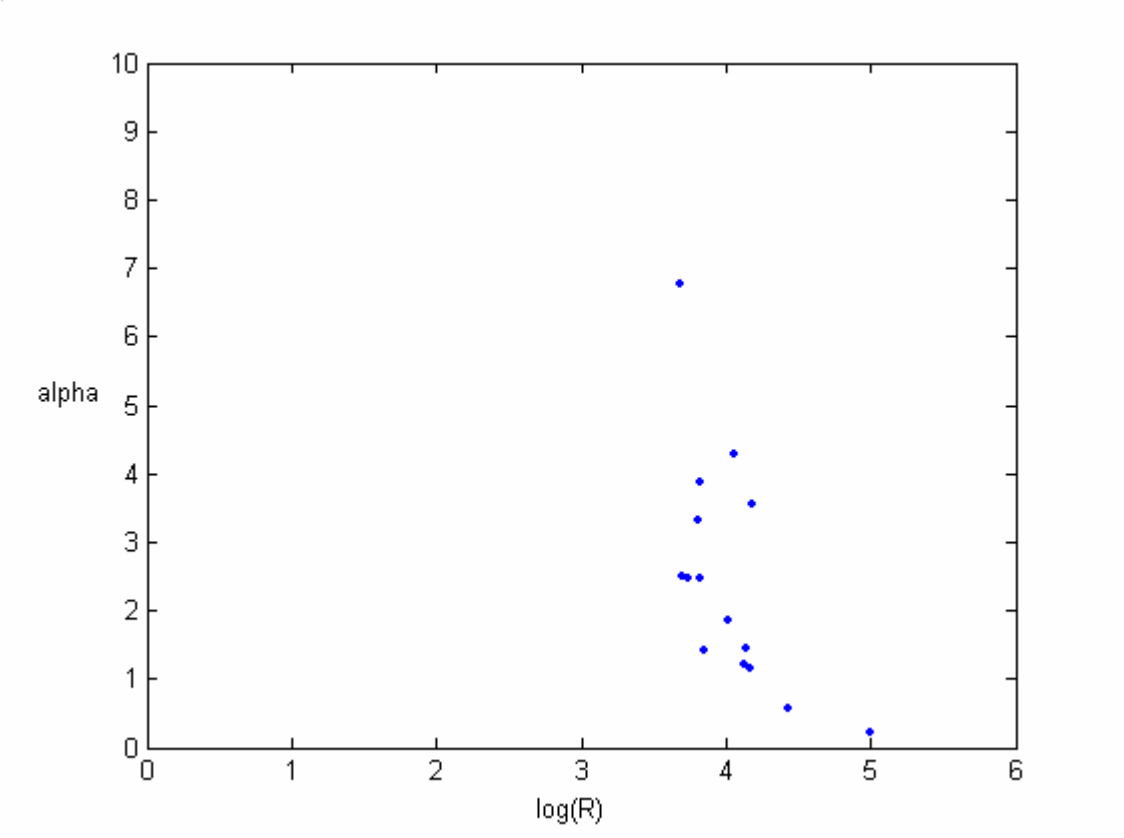


fig - 26

The primary reason for the deviation of points from a single smooth contour is the high oscillation of the solution $\phi_{II}(y)$ as the second numerical integration approaches $y = 0$. This fact is well demonstrated in the following plot of the amplitude of $\phi_{II}(y)$ vs. y for a particular α and R:

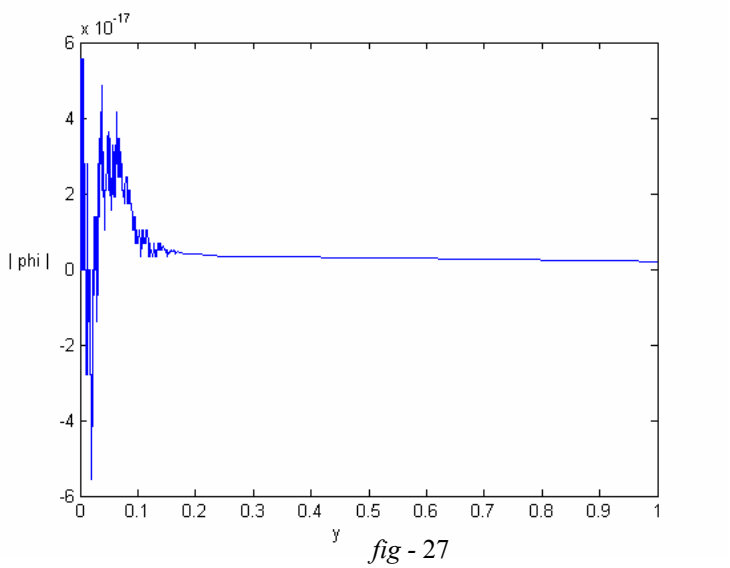


fig - 27

The small amplitude of $\phi_{II}(y)$ is due to the initial choice of a small A_2 for the second Integration pass.

4.4.3 Modified second Integration Pass using Stations in between:

This method, as explained by Betchov & Criminale [7], was implemented in order to reduce the oscillation of the second solution. The main principle of this method is based on the choice of some stations in between $y = 1$ and $y = 0$. At these stations the second integration is paused and it is updated by linearly combining with the first integration $\phi_1(y)$ to make $A_4 = 0$.

We are presently working on this technique and hope to obtain some satisfactory result very soon. Once we obtain a solution for the case of flow over a rigid, static plate, we'll attempt the solution for the case of flow over a flexible plate (modeled as an infinite beam) on the similar line.

4.5 Conclusions and discussions

Though a satisfactory numerical simulation could not be performed for studying the stability of flow over a flexible surface, the theory developed can have future applications. Proper numerical methods adopted for solving the Orr-Sommerfeld equation with the appropriate boundary conditions is expected to give a smooth $c_i = 0$ contour in the α-R plane, which in turn will give the critical Reynolds number that we are searching for.

5 Scope of future works

In future the dynamic model of the missile can be made more accurate by the following ways:

i) Inclusion of the yaw plane forces,
ii) The mode shapes and frequencies of a real life missile may be found experimentally or by FEM simulations and used instead of the modes of a free-free beam.
iii) Greater number of modes may be included.
iv) Better methods of determining the gain matrix $\boldsymbol{K}$ may be used to obtain more suitable values of the gain.
v) Effects of other types of forcing, including sloshing may be included.
vi) Higher order integration schemes will give better results
vii) The model once properly and completely developed, may be used as observer system with the real plant.

And as far as the study on stability of flow is concerned, as said before, a proper numerical solution couldn't be achieved. In future, with the use of better methods for solving the Orr-Sommerfeld equation with the appropriate boundary conditions, we may determine the critical Reynolds number for flows over flexible surfaces.

Appendix – I

Matlab Code for simulation of dynamics and control of the flexible missile

Below is given the primary M-files that were implemented. The function 'Solve_System' is the entry point to the code.

globals.m:

```
function globals
% ====================================================================
% Global variable declarations
% ====================================================================

% Control parameters
global del_p d_del_p dd_del_p   % Thrust angle in pitch plane
global del_y d_del_y dd_del_p   % Thrust angle in yaw plane
global Tc Ts                    % Thrust values

% Structural Constants
global materialE materialRho airRho
global m0 L0 csA meu
global xcg ycg zcg
global Ixx Iyy Izz Ixy Iyz Izx
global LC                   % Distance of the coordinate origin from hind end of the
       missile
global LR                   % Dist between engine cg and the hinge to which the
       engine is attached
global La                   % Distance between origine and nose tip of vehicle
global mR                   % Mass of engine
global mM                   % Mass of missile without engine
global I0 IR                % moment of inertia of engine abt cg and hinge
       respectively

% Net external Forces and Moments --> Time dependant scalers
global Fx Fy Fz
global Mx My Mz

% Angle, Velocity and accleration components --> constants
global U0 V0 W0
global P0 Q0 R0
global dU0 dV0 dW0
global dP0 dQ0 dR0
global pAng0 qAng0 rAng0
% Purturbation Angle, Velocity and accleration components --> Time dependant scalers
global pAng qAng rAng
global u v w
global p q r
global du dv dw
global dp dq dr

% Structural Variables
global modesNo          % No. of modes considered on each of pitch plane and yaw plane
global beta Amode Bmode % Mode shape definitions
% Pitch plane parameters
global zeta_p       % Damping coefficient
global omgMode_p    % Modal frequency for pitch plane modes
global q_p          % Coefficients of Mode shapes - 'modesNo' Time dependant scalers
global dq_p         % Time derivatives of q_p
global ddq_p        % 2nd time derivatives of q_p
global F_p          % Force per unit length in Pitch plane = force along Z axis -->
       Function of length
global M_p          % Moment per unit length in Pitch plane = moment along Y axis -->
       Function of length
global F_p_ddq_coef F_p_dq_coef F_p_q_coef % Global coeficients
global DC_p DC_p_ddq_coef DC_p_dq_coef DC_p_q_coef  % Derivative of moments due to
       couple along length
% Yaw plane parameters
global zeta_y       % Damping coefficient
global omgMode_y    % Modal frequency for yaw plane modes
global q_y          % Coefficients of Mode shapes - 'modesNo' Time dependant scalers
global dq_y         % Time derivatives of q_y
global ddq_y        % 2nd time derivatives of q_y
global F_y          % Force per unit length in Yaw plane = force along Y axis -->
       Function of length
global M_y          % Moment per unit length in Yaw plane = moment along Z axis -->
       Function of length
%
```

```
global F_a          % Force per unit length in axial direction = force along X axis --
       > Function of length
global M_a          % Moment per unit length due to twisting couples = moment along X
       axis --> Function of length

% Discreetization constants
global dt
global dl lSteps
global d_ddq_p

% Runtime variables
global time
global Tdata1 Tdata2

% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss rr                % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK uu Duu DDuu last_Duu      % Gain Matrix & Inputs for Integral Control
global MM_full NN_full      % For the full integral system
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated
global last_Duu             % Required for Integral control

% System and control types
global isTimeInvariant      % 1 => time invariant system --> reduces calculations
       considerably
global isIntegralControl    % 1 => uses Integral control --> for systems involving
       dd_del_p
global doFull               % 1 => solve the full integral system at a single time

% ====================================================================
% Global Variables values assignment
% ====================================================================

airRho = 1.2;
materialE = 7e10;               % Young's modulus
materialRho = 2600;             % Density
%materialE = 2e11;               % Young's modulus
%materialRho = 7800;             % Density
L0 = 1;                         % Length
csA = pi*0.1*0.1/4;             % C.S. area
mR = .1;                        % Engine mass
m0 = materialRho*L0*csA + mR;   % total mass
mM = m0 - mR;
meu = mM/L0;

Ixx = 2*pi*0.1*0.1*0.1*0.1*mM/csA;
Iyy = L0*L0*mM/12;
Izz = Iyy;
Ixy = 0;
Iyz = 0;
Izx = 0;
xcg = 0;    % Setting origin at the cg
ycg = 0;
zcg = 0;

LC = L0/2;  % Distance of the coordinate origin from hind end of the missile
LR = 0.04;  % Dist between engine cg and the hinge to which the engine is attached
La = L0 - LC;  % Distance between origine and nose tip of vehicle

I0 = mR*LC*LC; %100;   % engine moment of inertia about origine
IR = I0 + mR*LR*LR;

%%I_p = 0.5*(Ixx + Iyy - Izz);
I_p = pi*0.1*0.1*0.1*0.1/16;
I_y = I_p; %%0.5*(Ixx - Iyy + Izz);

modesNo = 10;
for n = 1:modesNo,
    zeta_p(n) = 0.01;
end

ctrlParmNo = 1;             % del_p is presently the only control parameter
stateVarNo = 9 + 2*modesNo;  % No of state variables % excluding sloshing
ExpressionVectorSize = 1 + 3*ctrlParmNo + 15 + 3*modesNo;

dt = 0.0000001; %1e-28;        % will depend max value of omega:  dt << 2*pi/omg_max
dl = 0.01;
lSteps = L0 / dl;
d_ddq_p = 0.000001;

% ====================================================================
% Global Variables values computation
% ====================================================================
```

```
% Finding betaL: Roots of cos(betaL)*cosh(betaL)=1
% 0th Root: 0
% 1nd root between 0.5*2pi and 1*2pi
% 2rd root between 1*2pi and 1.5*2pi
% 3th root between 1.5*2pi and 2*2pi
% ...
% nth root between (n)pi and (n+1)pi

% Calculating for 1st, 2nd and 3rd modes
for n = 1:3,
    lowBetaL = n*pi;
    hiBetaL = (n+1)*pi;
    lowVal = cos(lowBetaL)*cosh(lowBetaL) - 1;
    hiVal = cos(hiBetaL)*cosh(hiBetaL) - 1;
    lowSign = sign(lowVal);
    hiSign = sign(hiVal);
    while true,
        searchBetaL = (lowBetaL*hiVal - hiBetaL*lowVal)/(hiVal-lowVal);
        searchVal = cos(searchBetaL)*cosh(searchBetaL) - 1;
        if abs(searchVal) < 1e-10
            break;
        end
        searchSign = sign(searchVal);
        if searchSign == lowSign
            lowBetaL = searchBetaL;
            lowVal = searchVal;
        else
            lowBetaL = searchBetaL;
            hiVal = searchVal;
        end
    end
    beta(n) = searchBetaL/L0;
    Amode(n) = -(sinh(beta(n)*L0)-sin(beta(n)*L0));
    Bmode(n) = (cosh(beta(n)*L0)-cos(beta(n)*L0));
    omgMode_p(n) = beta(n)*beta(n)*sqrt((materialE*I_p)/(materialRho*csA));
    sprintf('Mode %d: beta=%1.10f , A=%d , B=%d, omg=%d', n, beta(n), Amode(n),
        Bmode(n), omgMode_p(n)),
end
% for n th mode with n>3, betaL = (n + 0.5)pi (approx.)
for n = 4:modesNo,
    beta(n) = (n + 0.5)*pi/L0;
    Amode(n) = -(sinh(beta(n)*L0)-sin(beta(n)*L0));
    Bmode(n) = (cosh(beta(n)*L0)-cos(beta(n)*L0));
    omgMode_p(n) = beta(n)*beta(n)*sqrt((materialE*I_p)/(materialRho*csA));
    sprintf('Mode %d: beta=%1.10f , A=%d , B=%d, omg=%d', n, beta(n), Amode(n),
        Bmode(n), omgMode_p(n)),
end
```

Solve_System.m:

```
function Solve_System
%
global time Tdata1 Tdata2
global solveTimeSteps
global dt L0 dl modesNo
global F_a F_y F_p M_a M_y M_p
% Global coeficients
global F_p_ddq_coef F_p_dq_coef F_p_q_coef
% Angle, Velocity and accleration components --> constants
global U0 V0 W0 dU0
global P0 Q0 R0
% Purturbation Angle, Velocity and accleration components --> Time dependant scalers
global pAng qAng rAng pAng0 qAng0 rAng0
global u v w
global p q r
global du dv dw
global dp dq dr
global del_p d_del_p dd_del_p   % Thrust angle in pitch plane
global del_y d_del_y dd_del_p   % Thrust angle in yaw plane
global Tc Ts                    % Thrust values
global q_p dq_p ddq_p q_y dq_y ddq_y        % Initial shape
global zeta_p omgMode_p
global DC_p DC_p_ddq_coef DC_p_dq_coef DC_p_q_coef  % Derivative of moments due to
       couple along length
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss rr                % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK uu Duu DDuu last_Duu      % Gain Matrix & Inputs for Integral Control
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated
```

```
global isTimeInvariant

warning off

isTimeInvariant = 1;
doFull = 1;

solveTimeSteps = 10000;
solveTime = solveTimeSteps*dt;      % Time for which solution to be performed

% Initilizing Values
globals
U0 = 1000;
dU0 = 0;
V0 = 0;
W0 = 0;
P0 = 0;
Q0 = 0;
R0 = 0;
pAng0 = 0;
qAng0 = 0.1;
rAng0 = 0;

% The global system equation is,
%           AA * Dss =  BB * ss  +  CC0 * uu  +  CC1 * Duu  +  CC2 * DDuu  +  DD
%
%  AA, BB -> stateVarNo x stateVarNo
%  CC0, CC1, CC2 -> stateVarNo x ctrlParmNo
%  DD, s -> stateVarNo x 1
%  u  -> ctrlParmNo x 1
%
% where ss is the state vector given by,
%           ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'
%
% and uu is the input vector of control parameters given by,
%           uu = [ cp(1)  cp(2)  ...  cp(ctrlParmNo) ]'
%
% Note: D = d/dt
%

ss = zeros(stateVarNo, 1);
rr = zeros(stateVarNo, 1);
ss(9+modesNo+1) = 1e-3;

uu = zeros(ctrlParmNo, 1);
Duu = zeros(ctrlParmNo, 1);
DDuu = zeros(ctrlParmNo, 1);
last_Duu = zeros(ctrlParmNo, 1);

Ts = 1e7;
Tc = 1e7;

% Initiating loop


% Looping
for time = 0:dt:solveTime,


    if isTimeInvariant==0 || time==0

        initiateSysEqn

        % ------------------------------------------------------------------


        % Force Calculation
        sprintf('Determining Forces .....')

        %force_calculate
        F_a = zeros(1+round(L0/dl), ExpressionVectorSize);
        F_p = zeros(1+round(L0/dl), ExpressionVectorSize);
        F_y = zeros(1+round(L0/dl), ExpressionVectorSize);
        M_a = zeros(1+round(L0/dl), ExpressionVectorSize);
        M_p = zeros(1+round(L0/dl), ExpressionVectorSize);
        M_y = zeros(1+round(L0/dl), ExpressionVectorSize);
        DC_p = zeros(1+round(L0/dl), ExpressionVectorSize);

        FMthrust;            % Due to Thrust
        FMinertia;           % Due to engine inertia
        FMaerodynamic;       % Aerodynamic force
        %FMgravity;
        % etc etc

        % ------------------------------------------------------------------
```

```
        % Adding equations to system Matrix
        sprintf('Adding equations due to structural flexibility .....')

        % Add structural equations due to flexibility
        structuralEqn_p;     % On pitch plane

        % Equations for from net Force and Moment balance
        OverallForceMomentEqn;

        % ------------------------------------------------------------------


        % Control loop for determining Gain KK
        sprintf('Determining gain matrix K .....')
        ControlSystem;
        % ....

        % ------------------------------------------------------------------


        % Checking the Open & Closed loop system for Stability, controllability, etc
        % Printing out the summery
        sprintf('Checking the system ..... \n     Plotting System Poles \n
       Printing system summery .....')
        SysCheck;

        %  ------------------------------------------------------------------

    end

    % Original time loop
    if mod(round(time/dt), 1) == 0
        sprintf('time = %d :   Determining input u ; Updating state vector s ; Saving
       Time Series Data .....', time)
    end
    % Control loop for updating input
    if doFull ~= 1
        ControlInput;
    end
    % Updates the state vector
    StateEval;

    % ------------------------------------------------------------------

    % Outputs:

    % Time Series data storing
    TimeSeriesProbes(time);

end

TimeSeriesPlots;
```

initiateSysEqn.m:

```
function initiateSysEqn
% initiates the System Equations -- defines extra system variables
%
% The global system equation is,
%           AA * Dss =  BB * ss  +  CC0 * uu  +  CC1 * Duu  +  CC2 * DDuu  +  DD
%
%  AA, BB -> stateVarNo x stateVarNo
%  CC0, CC1, CC2 -> stateVarNo x ctrlParmNo
%  DD, s -> stateVarNo x 1
%  u  -> ctrlParmNo x 1
%
% where ss is the state vector given by,
%           ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'
%
% and uu is the input vector of control parameters given by,
%           uu = [ cp(1)  cp(2)  ...  cp(ctrlParmNo) ]'
%
% Note: D = d/dt
%
%
% This function adds the equations
%           DpAng = dpAng/dt
%           DqAng = dqAng/dt
%           DrAng = drAng/dt
%           Dq_p(i) = dq_p(i)/dt    , i = 1 to modesNo
%

global modesNo
```

```
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss rr                % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK                   % Gain Matrix
global MM_full NN_full      % For the full integral system
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

AA = zeros(stateVarNo, stateVarNo);
BB = zeros(stateVarNo, stateVarNo);
CC0 = zeros(stateVarNo, ctrlParmNo);
CC1 = zeros(stateVarNo, ctrlParmNo);
CC2 = zeros(stateVarNo, ctrlParmNo);
DD = zeros(stateVarNo, 1);
KK = zeros(ctrlParmNo, stateVarNo);

LastEqnNo = 0;

% DpAng = dpAng/dt
% DqAng = dqAng/dt
% DrAng = drAng/dt
for a = 1:3,
    LastEqnNo = LastEqnNo + 1;
    AA(LastEqnNo, 6+a) = 1;
    BB(LastEqnNo, 3+a) = 1;
    CC0(LastEqnNo) = 0;
    CC1(LastEqnNo) = 0;
    CC2(LastEqnNo) = 0;
    DD(LastEqnNo) = 0;
end

% Dq_p(i) = dq_p(i)/dt    , i = 1 to modesNo
for a = 1:modesNo,
    LastEqnNo = LastEqnNo + 1;
    AA(LastEqnNo, 9+modesNo+a) = 1;
    BB(LastEqnNo, 9+a) = 1;
    CC0(LastEqnNo) = 0;
    CC1(LastEqnNo) = 0;
    CC2(LastEqnNo) = 0;
    DD(LastEqnNo) = 0;
end
```

FMaerodynamic.m:

```
function FMaerodynamic

global U0 W0 L0 dl airRho La L0 modesNo
global u q
global F_p M_p
global q_p
global F_p_ddq_coef F_p_dq_coef F_p_q_coef
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

% VV = [ independent_term
%        DDcp(1) DDcp(2) ... DDcp(ctrlParmNo)    Dcp(1) Dcp(2) ... Dcp(ctrlParmNo)
       cp(1) cp(2) ... cp(ctrlParmNo)
%        Du Dv Dw    u v w    DDpAng DDqAng DDrAng    DpAng DqAng DrAng    pAng qAng
       rAng
%        DDq_p(1) DDq_p(2) ... DDq_p(modesNo)   Dq_p(1)  Dq_p(2) ...  Dq_p(modesNo)
       q_p(1)  q_p(2) ...  q_p(modesNo) ]
%

alpha = atan(W0/U0);    % Angle of attack
A3 = 0.01;                % Reference area ?????

aeroF = zeros(1+round(L0/dl), ExpressionVectorSize);
for x = 0:dl:L0,
    indx = 1+round(x/dl);
    tmpCnAlpha = CnAlpha(x);
    aeroF(indx, 1) = aeroF(indx, 1) - 0.5*airRho*U0*U0*alpha*tmpCnAlpha;    % const
       term
    aeroF(indx, 1+3*ctrlParmNo+11) = aeroF(indx, 1+3*ctrlParmNo+11) -
       0.5*airRho*U0*tmpCnAlpha*(La-L0+x); % q
    aeroF(indx, 1+3*ctrlParmNo+4) = aeroF(indx, 1+3*ctrlParmNo+4) -
       0.5*airRho*2*U0*alpha*tmpCnAlpha - 0.5*airRho*tmpCnAlpha*(La-L0+x); % u
```

```
    for m = 1:modesNo,
        aeroF(indx, 1+3*ctrlParmNo+15+2*modesNo+m) = aeroF(indx,
       1+3*ctrlParmNo+15+2*modesNo+m) - 0.5*airRho*U0*U0*tmpCnAlpha*Dphi_p(m, x);
        aeroF(indx, 1+3*ctrlParmNo+15+modesNo+m) = aeroF(indx,
       1+3*ctrlParmNo+15+modesNo+m) - 0.5*airRho*U0*tmpCnAlpha*phi_p(m, x);
    end

    M_p(indx, :) = M_p(indx, :) - aeroF(indx, :)*(La-x);
end

F_p = F_p + aeroF;
```

CnAlpha.m:

```
function [CnAlpha] = CnAlpha(l)
% Pitch plane drag coeff
% dCn/dAlpha as a function of position along the vehicle, l
% Note: l = 0 at the tail of the missile
%       l = L0 at the tip

CnAlpha = 2*pi;
```

FMgravity.m:

```
function FMgravity

global U0 W0 L0 dl airRho La L0 modesNo
global u q xcg LC m0
global F_a F_p M_p
global q_p
global F_p_ddq_coef F_p_dq_coef F_p_q_coef
global pAng0 qAng0 rAng0
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

% VV = [ independent_term
%        DDcp(1) DDcp(2) ... DDcp(ctrlParmNo)    Dcp(1) Dcp(2) ... Dcp(ctrlParmNo)
       cp(1) cp(2) ... cp(ctrlParmNo)
%        Du Dv Dw    u v w    DDpAng DDqAng DDrAng    DpAng DqAng DrAng    pAng qAng
       rAng
%        DDq_p(1) DDq_p(2) ... DDq_p(modesNo)   Dq_p(1)  Dq_p(2) ...  Dq_p(modesNo)
       q_p(1)  q_p(2) ...  q_p(modesNo) ]
%

indx = 1+round((xcg+LC)/dl);
F_a(indx, 1+3*ctrlParmNo+14) = F_a(indx, 1+3*ctrlParmNo+14) + m0*9.8*sin(qAng0)/dl;
F_p(indx, 1+3*ctrlParmNo+14) = F_p(indx, 1+3*ctrlParmNo+14) - m0*9.8*cos(qAng0)/dl;
M_p(indx, 1+3*ctrlParmNo+14) = M_p(indx, 1+3*ctrlParmNo+14) + m0*(zcg*sin(qAng0) +
       xcg*cos(qAng0));
```

FMinertia.m:

```
function FMinertia
% Adds moments and forces due to engine inertia
% Presently only for pitch plane

global dl modesNo L0
global del_p d_del_p dd_del_p   % Thrust angle in pitch plane
global del_y d_del_y dd_del_p   % Thrust angle in yaw plane
global Tc Ts                    % Thrust values
global pAng qAng rAng
global u v w
global p q r
global du dv dw
global dp dq dr
global LC                   % Distance of the coordinate origin from hind end of the
       missile
global LR                   % Dist between engine cg and the hinge to which the
       engine is attached
global mR                   % Mass of engine
global I0 IR                % moment of inertia of engine abt cg and hinge
       respectively
global U0 dU0
global F_a F_p M_p DC_p
% State space formulation
```

```
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

% VV = [ independent_term
%        DDcp(1) DDcp(2) ... DDcp(ctrlParmNo)    Dcp(1) Dcp(2) ... Dcp(ctrlParmNo)
       cp(1) cp(2) ... cp(ctrlParmNo)
%        Du Dv Dw    u v w    DDpAng DDqAng DDrAng    DpAng DqAng DrAng    pAng qAng
       rAng
%        DDq_p(1) DDq_p(2) ... DDq_p(modesNo)   Dq_p(1)  Dq_p(2) ...  Dq_p(modesNo)
       q_p(1)  q_p(2) ...  q_p(modesNo) ]
%

F_a(1, 1+3*ctrlParmNo+1) = F_a(1, 1+3*ctrlParmNo+1) - mR/dl;

inertiaF = zeros(1+round(L0/dl), ExpressionVectorSize);
inertiaF(1, 2) = F_p(1, 2) - mR*LR;                                          %
       dd_del_p
inertiaF(1, 1+3*ctrlParmNo+7) = inertiaF(1, 1+3*ctrlParmNo+7) - mR*(LC+LR);    % dq
inertiaF(1, 1+3*ctrlParmNo+3) = inertiaF(1, 1+3*ctrlParmNo+3) - mR;            % dw
for m = 1:modesNo,
    inertiaF(1, 1+3*ctrlParmNo+15+m) = inertiaF(1, 1+3*ctrlParmNo+15+m) - mR*(phi_p(m,
        0) - LR*Dphi_p(m, 0));
end
F_p = F_p + inertiaF/dl;

inertiaC = zeros(1+round(L0/dl), ExpressionVectorSize);
inertiaC(1, 1+3*ctrlParmNo+3) = inertiaC(1, 1+3*ctrlParmNo+3) - mR*LR;          % dw
inertiaC(1, 1+3*ctrlParmNo+7) = inertiaC(1, 1+3*ctrlParmNo+7) - (mR*LR*LC+IR);  % dq
inertiaC(1, 2) = inertiaC(1, 2) - IR/(dl*dl);
       % dd_del_p
for m = 1:modesNo,
    inertiaC(1, 1+3*ctrlParmNo+15+m) = inertiaC(1, 1+3*ctrlParmNo+15+m) -
        (mR*LR*phi_p(m, 0) + IR*Dphi_p(m, 0));
end
DC_p = DC_p + inertiaC/(dl*dl);

M_p = M_p - inertiaC/dl + LC*inertiaF/dl;
```

FMthrust.m:

```
function FMthrust
% Adds moments and forces due to thrust
% Presently only for pitch plane
global dl LC modesNo
global del_p d_del_p dd_del_p   % Thrust angle in pitch plane
global del_y d_del_y dd_del_p   % Thrust angle in yaw plane
global Tc Ts                    % Thrust values
global F_p M_p DC_p
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

% VV = [ independent_term
%        DDcp(1) DDcp(2) ... DDcp(ctrlParmNo)    Dcp(1) Dcp(2) ... Dcp(ctrlParmNo)
       cp(1) cp(2) ... cp(ctrlParmNo)
%        Du Dv Dw    u v w    DDpAng DDqAng DDrAng    DpAng DqAng DrAng    pAng qAng
       rAng
%        DDq_p(1) DDq_p(2) ... DDq_p(modesNo)   Dq_p(1)  Dq_p(2) ...  Dq_p(modesNo)
       q_p(1)  q_p(2) ...  q_p(modesNo) ]
%

F_p(1, 1+2*ctrlParmNo+1) = F_p(1, 1+2*ctrlParmNo+1) +  Tc/dl;     % Coeff. of control
       parameter del_p
for m = 1:modesNo,
    F_p(1, 16+3*ctrlParmNo+2*modesNo+m) = F_p(1, 16+3*ctrlParmNo+2*modesNo+m) -
        (Tc+Ts)*Dphi_p(m, 0)/dl;
end

M_p(1, 1+2*ctrlParmNo+1) = M_p(1, 1+2*ctrlParmNo+1) +  LC*Tc/dl;     % Coeff. of
       control parameter del_p
for m = 1:modesNo,
    M_p(1, 16+3*ctrlParmNo+2*modesNo+m) = M_p(1, 16+3*ctrlParmNo+2*modesNo+m) -
        LC*(Tc+Ts)*Dphi_p(m, 0)/dl - (Tc+Ts)*phi_p(m, 0)/dl;
end
```

structuralEqn_p.m:

```
function structuralEqn_p
% first order precision
global L0 dl dt modesNo meu
global zeta_p       % Damping coefficient
global omgMode_p    % Modal frequency for pitch plane modes
global q_p          % Coefficients of Mode shapes - 'modesNo' Time dependant scalers
global dq_p         % Time derivatives of q_p
global ddq_p        % 2nd time derivatives of q_p
global F_p          % Force per unit length in Pitch plane = force along Z axis -->
       Function of length
global M_p          % Moment per unit length in Pitch plane = moment along Y axis -->
       Function of length
global DC_p
% Yaw plane parameters
global zeta_y       % Damping coefficient
global omgMode_y    % Modal frequency for yaw plane modes
global q_y          % Coefficients of Mode shapes - 'modesNo' Time dependant scalers
global dq_y         % Time derivatives of q_y
global ddq_y        % 2nd time derivatives of q_y
global F_y          % Force per unit length in Yaw plane = force along Y axis -->
       Function of length
global M_y          % Moment per unit length in Yaw plane = moment along Z axis -->
       Function of length
global time Tdata1 Tdata2

% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

% VV = [ independent_term
%        DDcp(1) DDcp(2) ... DDcp(ctrlParmNo)    Dcp(1) Dcp(2) ... Dcp(ctrlParmNo)
       cp(1) cp(2) ... cp(ctrlParmNo)
%        Du Dv Dw    u v w    DDpAng DDqAng DDrAng    DpAng DqAng DrAng    pAng qAng
       rAng
%        DDq_p(1) DDq_p(2) ... DDq_p(modesNo)   Dq_p(1)  Dq_p(2) ...  Dq_p(modesNo)
       q_p(1)  q_p(2) ...  q_p(modesNo) ]
%

for n = 1:modesNo, % looping over n equations

    genM = 0;
    for m = 1:ExpressionVectorSize,
        genQ(m) = 0;
    end
    for x = 0:dl:L0,
        tmpPhi = phi_p(n, x);
        for m = 1:ExpressionVectorSize,
            genQ(m) = genQ(m) + (F_p(1+round(x/dl), m)-
        DC_p(1+round(x/dl)))*tmpPhi*dl;
        end
        genM = genM + meu*tmpPhi*tmpPhi*dl;
    end
    genQbyM = genQ/genM;

    thisEqnVector = -genQbyM;
    thisEqnVector(16+3*ctrlParmNo+n) = thisEqnVector(16+3*ctrlParmNo+n) + 1.0;
    thisEqnVector(16+3*ctrlParmNo+modesNo+n) =
        thisEqnVector(16+3*ctrlParmNo+modesNo+n) + 2*zeta_p(n)*omgMode_p(n);
    thisEqnVector(16+3*ctrlParmNo+2*modesNo+n) =
        thisEqnVector(16+3*ctrlParmNo+2*modesNo+n) + omgMode_p(n)*omgMode_p(n);

    generateEquation(thisEqnVector);

end
```

OverallForceMomentEqn.m:

```
function OverallForceMomentEqn
% Update the linear and angular acleration terms using the
% net external Forces and Moments equation
global dl L0 m0 U0 dU0 dt mM qAng0 Q0 dQ0
global xcg ycg zcg
global Ixx Iyy Izz Ixy Iyz Izx
global pAng qAng rAng
global u v w
global p q r
```

```
global du dv dw
global dp dq dr
global F_a F_y F_p M_a M_y M_p
global modesNo

% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

% VV = [ independent_term
%        DDcp(1) DDcp(2) ... DDcp(ctrlParmNo)    Dcp(1) Dcp(2) ... Dcp(ctrlParmNo)
       cp(1) cp(2) ... cp(ctrlParmNo)
%        Du Dv Dw    u v w    DDpAng DDqAng DDrAng    DpAng DqAng DrAng    pAng qAng
       rAng
%        DDq_p(1) DDq_p(2) ... DDq_p(modesNo)   Dq_p(1)  Dq_p(2) ...  Dq_p(modesNo)
       q_p(1)  q_p(2) ...  q_p(modesNo) ]
%

% Calculating overall force and moments from pitch and yaw plane forces and moments.
for m = 1:ExpressionVectorSize,
    Fx(m) = 0;
    Fy(m) = 0;
    Fz(m) = 0;
    Mx(m) = 0;
    My(m) = 0;
    Mz(m) = 0;
end
for x = 0:dl:L0,
    indx = 1+round(x/dl);
    for m = 1:ExpressionVectorSize,
        Fx(m) = Fx(m) + F_a(indx, m)*dl;
        Fy(m) = Fy(m) + F_y(indx, m)*dl;
        Fz(m) = Fz(m) + F_p(indx, m)*dl;
        My(m) = My(m) + M_p(indx, m)*dl; % - F_p(1+x/dl)*(x-LC)*dl;
        Mz(m) = Mz(m) + M_y(indx, m)*dl; % + F_y(1+x/dl)*(x-LC)*dl;
        Mx(m) = Mx(m) + M_a(indx, m)*dl;
    end
end

% Equations 45 to 50

thisTmpEqn = -Fx;
thisTmpEqn(1+3*ctrlParmNo+1) = thisTmpEqn(1+3*ctrlParmNo+1) + mM;
thisTmpEqn(1+3*ctrlParmNo+9) = thisTmpEqn(1+3*ctrlParmNo+9) - mM*ycg;
thisTmpEqn(1+3*ctrlParmNo+8) = thisTmpEqn(1+3*ctrlParmNo+8) + mM*zcg;
generateEquation(thisTmpEqn);

thisTmpEqn = -Fy;
thisTmpEqn(1+3*ctrlParmNo+2) = thisTmpEqn(1+3*ctrlParmNo+2) + mM;
thisTmpEqn(1+3*ctrlParmNo+12) = thisTmpEqn(1+3*ctrlParmNo+12) + mM*U0;
thisTmpEqn(1+3*ctrlParmNo+9) = thisTmpEqn(1+3*ctrlParmNo+9) + mM*xcg;
thisTmpEqn(1+3*ctrlParmNo+7) = thisTmpEqn(1+3*ctrlParmNo+7) - mM*zcg;
generateEquation(thisTmpEqn);

thisTmpEqn = -Fz;
thisTmpEqn(1+3*ctrlParmNo+3) = thisTmpEqn(1+3*ctrlParmNo+3) + mM;
thisTmpEqn(1+3*ctrlParmNo+11) = thisTmpEqn(1+3*ctrlParmNo+11) - mM*U0;
thisTmpEqn(1+3*ctrlParmNo+8) = thisTmpEqn(1+3*ctrlParmNo+8) - mM*xcg;
thisTmpEqn(1+3*ctrlParmNo+7) = thisTmpEqn(1+3*ctrlParmNo+7) + mM*ycg;
generateEquation(thisTmpEqn);

thisTmpEqn = -Mx;
thisTmpEqn(1+3*ctrlParmNo+7) = thisTmpEqn(1+3*ctrlParmNo+7) + Ixx;
thisTmpEqn(1+3*ctrlParmNo+8) = thisTmpEqn(1+3*ctrlParmNo+8) - Ixy;
thisTmpEqn(1+3*ctrlParmNo+9) = thisTmpEqn(1+3*ctrlParmNo+9) - Izx;
thisTmpEqn(1+3*ctrlParmNo+11) = thisTmpEqn(1+3*ctrlParmNo+11) - U0*mM*ycg;
thisTmpEqn(1+3*ctrlParmNo+12) = thisTmpEqn(1+3*ctrlParmNo+12) - U0*mM*zcg;
thisTmpEqn(1+3*ctrlParmNo+2) = thisTmpEqn(1+3*ctrlParmNo+2) - mM*zcg;
thisTmpEqn(1+3*ctrlParmNo+3) = thisTmpEqn(1+3*ctrlParmNo+3) + mM*ycg;
generateEquation(thisTmpEqn);

thisTmpEqn = -My;
thisTmpEqn(1+3*ctrlParmNo+7) = thisTmpEqn(1+3*ctrlParmNo+7) - Ixy;
thisTmpEqn(1+3*ctrlParmNo+8) = thisTmpEqn(1+3*ctrlParmNo+8) + Iyy;
thisTmpEqn(1+3*ctrlParmNo+9) = thisTmpEqn(1+3*ctrlParmNo+9) - Iyz;
thisTmpEqn(1+3*ctrlParmNo+11) = thisTmpEqn(1+3*ctrlParmNo+11) + U0*mM*xcg;
thisTmpEqn(1+3*ctrlParmNo+1) = thisTmpEqn(1+3*ctrlParmNo+1) + mM*zcg;
thisTmpEqn(1+3*ctrlParmNo+3) = thisTmpEqn(1+3*ctrlParmNo+3) + mM*xcg;
generateEquation(thisTmpEqn);

thisTmpEqn = -My;
thisTmpEqn(1+3*ctrlParmNo+7) = thisTmpEqn(1+3*ctrlParmNo+7) - Izx;
```

```
thisTmpEqn(1+3*ctrlParmNo+8) = thisTmpEqn(1+3*ctrlParmNo+8) - Iyz;
thisTmpEqn(1+3*ctrlParmNo+9) = thisTmpEqn(1+3*ctrlParmNo+9) + Izz;
thisTmpEqn(1+3*ctrlParmNo+12) = thisTmpEqn(1+3*ctrlParmNo+12) + U0*mM*xcg;
thisTmpEqn(1+3*ctrlParmNo+1) = thisTmpEqn(1+3*ctrlParmNo+1) - mM*ycg;
thisTmpEqn(1+3*ctrlParmNo+2) = thisTmpEqn(1+3*ctrlParmNo+2) + mM*xcg;
generateEquation(thisTmpEqn);
```

SysCheck.m:

```
function SysCheck
%
% The global system equation is,
%           AA * Dss =  BB * ss  +  CC0 * uu  +  CC1 * Duu  +  CC2 * DDuu  +  DD
%
%  AA, BB -> stateVarNo x stateVarNo
%  CC0, CC1, CC2 -> stateVarNo x ctrlParmNo
%  DD, s -> stateVarNo x 1
%  u  -> ctrlParmNo x 1
%
% where ss is the state vector given by,
%           ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'
%
% and uu is the input vector of control parameters given by,
%           uu = [ cp(1)  cp(2)  ...  cp(ctrlParmNo) ]'
%
% Note: D = d/dt
%

global modesNo
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global MM_full NN_full      % For the full integral system
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated
global uu last_uu last2last_uu Duu DDuu KK  % Control loop parameters & inputs


sprintf('System properties: \n     System equation: A*d(ss) = B*ss + C0*uu + C1*d(uu)
       + C2*dd(uu) + D \n     Integral Control: d(uu) = K*ss \n     [Note: d = d/dt]')


sprintf('Poles of Open Loop system \n i.e. Eigenvalues of inv(A)*B .....')
Rmat = inv(AA)*BB;
eigenVals = eig(Rmat);
eigenVals,
hold on;
figure;
plot(eigenVals, '.');
title('Poles of Open Loop system');
hold off;

sprintf('Poles of Closed Loop system with Integral Control ..... \n i.e. Eigenvalues
       of \n        --                                                          --
       \n        ||                                |                              ||\n
       || inv(A-C2*K)*(B+C1*K) - lamb*I  |     inv(A-C2*K)*C0*K          ||\n        ||
       |                                 ||\n        ||--------------------------------|-
       ------------------------------||      \n        ||
       |                                 ||\n        ||             I                 |
       -lamb*I            ||\n        ||                                |
       ||\n        --                                                          --
       .....')
sprintf('With zero gain .....')
zero_KK = zeros(ctrlParmNo, stateVarNo);
this_KK = zero_KK;
tmpMat = inv(AA-CC2*this_KK);
MM11 = tmpMat*(BB+CC1*this_KK);
MM12 = tmpMat*CC0*this_KK;
MM21 = eye(stateVarNo);
MM22 = zeros(stateVarNo);
MM = [ MM11 MM12 ; MM21 MM22 ];
eigenVals = eig(MM);
eigenVals,
hold on;
figure;
plot(eigenVals, '.');
title('Poles of Closed Loop system with zero gain');
hold off;

if KK == zero_KK
    sprintf('Gain determined to stabilize Integral feedback loop is ZERO!!!!!')
```

```
else
    sprintf('With gain determined to stabilize Integral feedback loop =')
    KK,
    eigenVals = eig(MM_full);
    eigenVals,
    hold on;
    figure;
    plot(eigenVals, '.');
    title('Poles of Closed Loop system with gain determined to stabilize Integral
       feedback loop');
    hold off;
end


% Saving the system Matrices
save 'last_run_system.mat' AA BB CC0 CC1 CC2 DD KK

pause;
```

ControlSystem.m:

```
function ControlSystem

% The global system equation is,
%           AA * Dss =  BB * ss  +  CC0 * uu  +  CC1 * Duu  +  CC2 * DDuu  +  DD
%
%  AA, BB -> stateVarNo x stateVarNo
%  CC0, CC1, CC2 -> stateVarNo x ctrlParmNo
%  DD, s -> stateVarNo x 1
%  u  -> ctrlParmNo x 1
%
% where ss is the state vector given by,
%           ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'
%
% and uu is the input vector of control parameters given by,
%           uu = [ cp(1)  cp(2)  ...  cp(ctrlParmNo) ]'
%
% Note: D = d/dt
%

% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK                   % Gain Matrix
global MM_full NN_full      % For the full integral system
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

global eigVals1

Rmat = inv(AA)*BB;
eigVals = eig(Rmat);
ord=.1;
thePoles = eig(inv(AA)*BB)*5 + (-(0.7+0.3*rand(stateVarNo, 1))*2 +
       (10*rand(stateVarNo, 1))*i)*ord;
sprintf('Placing poles of the system for Integral Control ..... \n  Determining K by
       Newton-Raphson iteration ....')
PolePlaceIgame;
%KK(2) = 9.600621147974315e+024;

tmpMat = inv(AA-CC2*KK);
MM11 = tmpMat*(BB+CC1*KK);
MM12 = tmpMat*CC0*KK;
MM21 = eye(stateVarNo);
MM22 = zeros(stateVarNo);
MM_full = [ MM11 MM12 ; MM21 MM22 ];    % / (0.5e100/stateVarNo);

NN1 = tmpMat*DD;
NN2 = zeros(stateVarNo, 1);
NN_full = [ NN1 ; NN2 ];
```

ControlInput.m:

```
function ControlInput

% The global system equation is,
%           AA * Dss =  BB * ss  +  CC0 * uu  +  CC1 * Duu  +  CC2 * DDuu  +  DD
%
%  AA, BB -> stateVarNo x stateVarNo
```

```
%  CC0, CC1, CC2 -> stateVarNo x ctrlParmNo
%  DD, s -> stateVarNo x 1
%  u  -> ctrlParmNo x 1
%
% where ss is the state vector given by,
%           ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'
%
% and uu is the input vector of control parameters given by,
%           uu = [ cp(1)  cp(2)  ...  cp(ctrlParmNo) ]'
%
% Note: D = d/dt
%

global dt
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK uu Duu DDuu last_Duu      % Gain Matrix & Inputs for Integral Control
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

% -+- Generating UU -+-
% Integral control
last_Duu = Duu;
Duu = KK*ss;
uu = uu + Duu*dt;
DDuu = (Duu - last_Duu)/dt;
```

phi_p.m:

```
function [phi_p] = phi_p(n, x)
% Returns the displacement of n th mode shape at position x
global beta Amode Bmode

betaX = beta(n)*x;
phi_p = Amode(n)*(cosh(betaX) + cos(betaX)) + Bmode(n)*(sinh(betaX) + sin(betaX));
```

Dphi_p.m:

```
function [Dphi_p] = Dphi_p(n, x)
% Returns the slope of n th mode shape at position x
global beta Amode Bmode

betaX = beta(n)*x;
Dphi_p = beta(n) * (Amode(n)*(sinh(betaX) - sin(betaX)) + Bmode(n)*(cosh(betaX) +
       cos(betaX)));
```

generateEquation.m:

```
function generateEquation(eqnVector)
% adds an equation to the global state-space matrices
%
% the equation is,
%           dot(eqnVector, VV) = 0
% where,
% VV = [ independent_term
%        DDcp(1) DDcp(2) ... DDcp(ctrlParmNo)    Dcp(1) Dcp(2) ... Dcp(ctrlParmNo)
       cp(1) cp(2) ... cp(ctrlParmNo)
%        Du Dv Dw    u v w    DDpAng DDqAng DDrAng    DpAng DqAng DrAng    pAng qAng
       rAng
%        DDq_p(1) DDq_p(2) ... DDq_p(modesNo)   Dq_p(1)  Dq_p(2) ...  Dq_p(modesNo)
       q_p(1)  q_p(2) ...  q_p(modesNo) ]
%
% The global system equation is,
%           AA * Dss =  BB * ss  +  CC0 * uu  +  CC1 * Duu  +  CC2 * DDuu  +  DD
%
%  AA, BB -> stateVarNo x stateVarNo
%  CC0, CC1, CC2 -> stateVarNo x ctrlParmNo
%  DD, s -> stateVarNo x 1
%  u  -> ctrlParmNo x 1
%
% where ss is the state vector given by,
%           ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'
%
% and uu is the input vector of control parameters given by,
%           uu = [ cp(1)  cp(2)  ...  cp(ctrlParmNo) ]'
```

```
%
% Note: D = d/dt
%

global modesNo
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

LastEqnNo = LastEqnNo + 1;

% u v w
for a = 1:3,
    AA(LastEqnNo, a) = eqnVector(1+3*ctrlParmNo+a);
    BB(LastEqnNo, a) = -eqnVector(4+3*ctrlParmNo+a);
end

% pAng qAng rAng
for a = 1:3,
    AA(LastEqnNo, 3+a) = eqnVector(7+3*ctrlParmNo+a);
end
for a = 1:6,
    BB(LastEqnNo, 3+a) = -eqnVector(10+3*ctrlParmNo+a);
end

% q_p(i), i = 1 to modesNo
for a = 1:modesNo,
    AA(LastEqnNo, 9+a) = eqnVector(16+3*ctrlParmNo+a);
end
for a = 1:2*modesNo,
    BB(LastEqnNo, 9+a) = -eqnVector(16+3*ctrlParmNo+modesNo+a);
end

% Control parameters
for a = 1:ctrlParmNo,
    CC0(LastEqnNo) = -eqnVector(1+2*ctrlParmNo+a);
    CC1(LastEqnNo) = -eqnVector(1+ctrlParmNo+a);
    CC2(LastEqnNo) = -eqnVector(1+a);
end
% const term
DD(LastEqnNo) = -eqnVector(1);


% Normalizing
AAfactor = max(abs(AA(LastEqnNo,:)));
AA(LastEqnNo,:) = AA(LastEqnNo,:) / AAfactor;
BB(LastEqnNo,:) = BB(LastEqnNo,:) / AAfactor;
CC0(LastEqnNo,:) = CC0(LastEqnNo,:) / AAfactor;
CC1(LastEqnNo,:) = CC1(LastEqnNo,:) / AAfactor;
CC2(LastEqnNo,:) = CC2(LastEqnNo,:) / AAfactor;
DD(LastEqnNo,:) = DD(LastEqnNo,:) / AAfactor;
```

PolePlaceIgame.m:

```
function PolePlaceIgame

%
% Places the poles for Integral feedback control on the left of imaginary axis using
       genetic algorithm
%
% Integral control =>
%           uu = KK * Integration[ss * dt]
%
% Can be used when C2 is not zero.
%
% ------------------------------------------------------------------------
% Given the position of the poles, 'lamb', in the cplx plane, determines the gain
       matrix K
% K = [k1 k2 k3 ... kn]
%
% The equation to be satisfied is,
%
%         --                                                          --
%        ||                                  |                         ||
%        ||  inv(A-C2*K)*(B+C1*K) - lamb*I   |     inv(A-C2*K)*C0*K    ||
%        ||                                  |                         ||
%    det ||----------------------------------|-------------------------||   =  0
%        ||                                  |                         ||
%        ||              I                   |            -lamb*I      ||
%        ||                                  |                         ||
```

```
%         --                                                              --
%
% There are 2n poles (i.e. 2n 'lamb's), which gives 2n equations.
% But if we assume that A, B, C0, C1, C2 and K have only real elements,
% 'lamb' will consist of complex conjugates.
% Hence, lamb has n elements.
% where n = stateVarNo

% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK                   % Gain Matrix

tmp_KK = zeros(1, stateVarNo);

maxIter=1000;
for a=1:maxIter,
    isThisKKok = isThisKKokay(tmp_KK);
    if isThisKKok==1
        break;
    elseif isThisKKok==-1
        sprintf('WARNING!! Divergence in iteration no.: %d -- Scaling K by 1e-10!!',
       a)
        tmp_KK = tmp_KK * 1e-10;
    end

    % Finding best scores and corrosponding changes of the present set of K
    KK_score = zeros(1, stateVarNo);
    delta_KK = zeros(1, stateVarNo);
    for b=1:stateVarNo,
        [delta, score] = findBestDelta3(tmp_KK, b);
        KK_score(b) = score;
        delta_KK(b) = delta;
        sprintf('turn number=%d , K index=%d : score=%d , delta=%d', a, b, score,
       delta)
    end

    [max_score, winner_index] = max(KK_score);
    tmp_KK(winner_index) = tmp_KK(winner_index) + delta_KK(winner_index);

    sprintf('No solution till turn number %d :  Last winner score = %d', a, max_score)
end


if a==maxIter
    sprintf('WARNING: No solution for KK found!!!')
else
    sprintf('*** Solution in turn number %d', a)
    KK = tmp_KK;
end


% ----------------------------
%       End of Function
% ============================


function [g]=g(this_KK)

%
% given a particular K and lamb, this function calculates the vector g
%
%
%                --                                          --
%               ||                          |                 ||
%               ||  inv(A-C2*K)*(B+C1*K)    |    inv(A-C2*K)*C0*K  ||
%               ||                          |                 ||
%    g =   eig  ||--------------------------|-----------------||
%               ||                          |                 ||
%               ||            I             |         0       ||
%               ||                          |                 ||
%                --                                          --
%


% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK                   % Gain Matrix
```

```
global LastEqnNo            % Variable to keep track of the number of equation
      incorporated
global lamb


%for n = 1:stateVarNo,
    tmpMat = inv(AA-CC2*this_KK);
    MM11 = tmpMat*(BB+CC1*this_KK);
    MM12 = tmpMat*CC0*this_KK;
    MM21 = eye(stateVarNo);
    MM22 = zeros(stateVarNo);
    MM = [ MM11 MM12 ; MM21 MM22 ];    % / (0.5e100/stateVarNo);
    eigenVals = eig(MM);
    g = sort(real(eigenVals));



% ----------------------------
%       End of Function
% ============================


function  [delta, score] = findBestDelta3(this_KK, Kindex)
% search for highest score by multiple partition search

no_of_points = 1000;
delta_max = 1e25;
delta_min = -1e25;
delta_gap = (delta_max-delta_min)/(no_of_points+1);

delta_points = delta_min:delta_gap:delta_max;

for a=1:6,
    for b=1:no_of_points,
        score_points(b) = getScore(this_KK, Kindex, delta_points(b));
    end
    [max_score, max_index] = max(score_points);
    the_best_delta = delta_points(max_index);
    delta_max = the_best_delta+delta_gap;
    delta_min = the_best_delta-delta_gap;
    delta_gap = 2*delta_gap/(no_of_points+1);
    delta_points = delta_min:delta_gap:delta_max;
end

delta = the_best_delta;
score = getScore(this_KK, Kindex, delta);

% ----------------------------
%       End of Function
% ============================


function [score] = getScore(this_KK, Kindex, delta)
global stateVarNo

% Parameters in calculating the score
gain_in_positive = 1e-3;
gain_in_negetive = 0.5*1e-3;
gain_in_cross = 1;
% ----------------------------------

tempKK = this_KK;
old_g = g(tempKK);
tempKK(Kindex) = tempKK(Kindex) + delta;
new_g = g(tempKK);
delta_g = old_g - new_g;
gain_in_cross = gain_in_cross*max(abs(delta_g));

score=0;
for a=1:2*stateVarNo,
    old_sign = sign(old_g(a));
    new_sign = sign(new_g(a));
    if old_sign ~= new_sign
        if old_sign == 1
            score = score + gain_in_cross;
        elseif new_sign == 1
            score = score - gain_in_cross;
        end
    end
    if new_sign == 1
        score = score + gain_in_positive*delta_g(a);
    else
        score = score + gain_in_negetive*delta_g(a);
    end
end

score = score / gain_in_cross;
```

```
% ----------------------------
%       End of Function
% ============================



function isThisKKokay = isThisKKokay(this_KK)

%
% given a particular K and lamb, this function calculates the vector g
%
%
%               --                                                --
%              ||                                                  ||
%              ||  inv(A-C2*K)*(B+C1*K)    |     inv(A-C2*K)*C0*K  ||
%              ||                          |                       ||
%    g =   eig ||--------------------------|-----------------------||
%              ||                          |                       ||
%              ||             I            |          0            ||
%              ||                          |                       ||
%               --                                                --
%


% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss                   % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK                   % Gain Matrix
global LastEqnNo            % Variable to keep track of the number of equation
      incorporated
global lamb


%for n = 1:stateVarNo,
    tmpMat = inv(AA-CC2*this_KK);
    MM11 = tmpMat*(BB+CC1*this_KK);
    MM12 = tmpMat*CC0*this_KK;
    MM21 = eye(stateVarNo);
    MM22 = zeros(stateVarNo);
    MM = [ MM11 MM12 ; MM21 MM22 ];    % / (0.5e100/stateVarNo);
    try
        eigenVals = eig(MM);
    catch
        isThisKKokay = -1;
        return;
    end

    isThisKKokay = 1;
    for n = 1:2*stateVarNo,
        if real(eigenVals(n)) > 0
            isThisKKokay = 0;
        end
    end

% ----------------------------
%       End of Function
% ============================
```

StateEval.m:

```
function StateEval
% Evaluates the state vector

% The global system equation is,
%           AA * Dss =  BB * ss  +  CC0 * uu  +  CC1 * Duu  +  CC2 * DDuu  +  DD
%
%  AA, BB -> stateVarNo x stateVarNo
%  CC0, CC1, CC2 -> stateVarNo x ctrlParmNo
%  DD, s -> stateVarNo x 1
%  u  -> ctrlParmNo x 1
%
% where ss is the state vector given by,
%           ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'
%
% and uu is the input vector of control parameters given by,
%           uu = [ cp(1)  cp(2)  ...  cp(ctrlParmNo) ]'
%
% Note: D = d/dt
```

```
%

global dt doFull
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss rr                % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK
global MM_full NN_full      % For the full integral system
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated
global del_p d_del_p dd_del_p   % Thrust angle in pitch plane
global uu last_uu last2last_uu Duu DDuu KK  % Control loop parameters & inputs


% Updating state vector
if doFull ~= 1
    AAinv = inv(AA);
    ss = ss  +  dt * AAinv*(BB*ss + CC0*uu + CC1*Duu + CC2*DDuu + DD);
else
    ss_rr = [ ss ; rr ];
    sub_steps = 1000;
    sub_dt = dt/sub_steps;
    for a=1:sub_steps,
        ss_rr = ss_rr  +  sub_dt * (MM_full*ss_rr + NN_full);
    end
    ss = ss_rr(1:stateVarNo);
    rr = ss_rr(1+stateVarNo:2*stateVarNo);
end
```

shapeDisp_p.m:

```
function [shapeDisp_p] = shapeDisp_p(l)
% Returns the current shape displacement in pitch plane at position l
%
% ss, the state vector given by,
%       ss = [ u v w   DpAng DqAng DrAng   pAng qAng rAng   Dq_p(1) Dq_p(2) ...
       Dq_p(modesNo)   q_p(1) q_p(2) ... q_p(modesNo)]'

global q_p modesNo
global ss

shapeDisp_p = 0;
for n = 1:modesNo,
    shapeDisp_p = shapeDisp_p + ss(9+modesNo+n)*phi_p(n, l);
end
```

TimeSeriesProbes.m:

```
function TimeSeriesProbes(this_time)

% Data for Missile shape
    global Data1TimePointsNo
    global title1
    global TData1
    global TVariable1
% Data for Displacement of hind end of missile
    global Data2TimePointsNo
    global title2
    global TData2
    global TVariable2

% Global data for including in time series data
global dt L0 solveTimeSteps ss U0

%---------------------------------------
timeIndex = round(this_time/dt) + 1;
%---------------------------------------

% Shape of the Missile
Data1TimePointsNo = 10;                                          % Change
TimeStepInterval = round(solveTimeSteps/Data1TimePointsNo);      % Change
if mod(timeIndex, TimeStepInterval) == 0
    dataIndex = round(timeIndex/TimeStepInterval);
    title1 = 'Shape of the missile at different time instants'; % Change
    TVariable1 = (0:L0/100:L0);                                  % Change
    TData1(dataIndex, :) = shapeDisp_p(TVariable1);              % Change
end

% Displacement of hind end of missile
Data2TimePointsNo = 1000;
```

```
TimeStepInterval = round(solveTimeSteps/Data2TimePointsNo);
if mod(timeIndex, TimeStepInterval) == 0
    dataIndex = round(timeIndex/TimeStepInterval);
    title2 = 'Displacement at different positions with time';
    TVariable2(dataIndex) = this_time;
    TData2(1, dataIndex) = shapeDisp_p(L0);
    TData2(2, dataIndex) = shapeDisp_p(L0/2);
    TData2(3, dataIndex) = shapeDisp_p(0);
    %shapeDisp_p(0),
end
```

TimeSeriesPlots.m:

```
function TimeSeriesPlots

% Data for Missile shape
    global title1
    global TData1
    global TVariable1
% Data for Displacement of hind end of missile
    global title2
    global TData2
    global TVariable2

hold on;
figure;
plot(TVariable1, TData1);
title(title1);
hold off;

hold on;
figure;
size(TData2),
plot(TVariable2, TData2(1,:), 'r-', TVariable2, TData2(2,:), 'b:', TVariable2,
       TData2(3,:), 'k-.');
title(title2);
legend('at x=L0','at x=L0/2','at x=0');
hold off;


% Saving the outputs
save 'last_run_outputs.mat' TData1 TVariable1 TData2 TVariable2
```

RootLocusPlot.m:

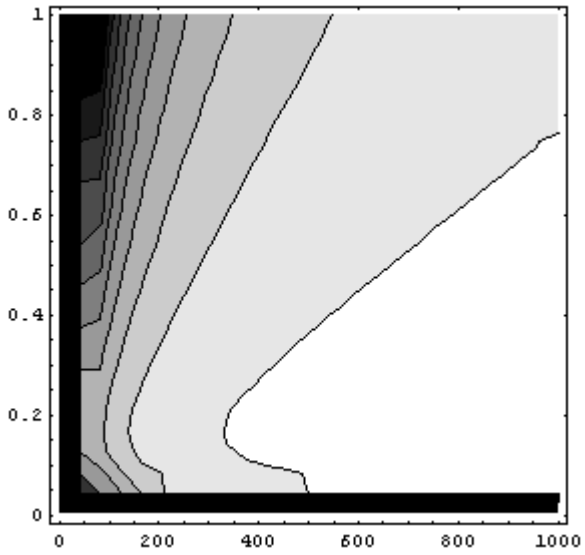
```
function RootLocusPlot

%
global time Tdata1 Tdata2
global solveTimeSteps
global dt L0 dl modesNo
global F_a F_y F_p M_a M_y M_p
% Global coeficients
global F_p_ddq_coef F_p_dq_coef F_p_q_coef
% Angle, Velocity and accleration components --> constants
global U0 V0 W0 dU0
global P0 Q0 R0
% Purturbation Angle, Velocity and accleration components --> Time dependant scalers
global pAng qAng rAng pAng0 qAng0 rAng0
global u v w
global p q r
global du dv dw
global dp dq dr
global del_p d_del_p dd_del_p   % Thrust angle in pitch plane
global del_y d_del_y dd_del_p   % Thrust angle in yaw plane
global Tc Ts                    % Thrust values
global q_p dq_p ddq_p q_y dq_y ddq_y        % Initial shape
global zeta_p omgMode_p
global DC_p DC_p_ddq_coef DC_p_dq_coef DC_p_q_coef  % Derivative of moments due to
       couple along length
% State space formulation
global stateVarNo           % No of state variables
global ctrlParmNo           % No of control parameters
global ExpressionVectorSize % No of elements in an 'expression vector'
global ss rr                % State vector
global AA BB CC0 CC1 CC2 DD % System equation matrices
global KK uu Duu DDuu last_Duu      % Gain Matrix & Inputs for Integral Control
global LastEqnNo            % Variable to keep track of the number of equation
       incorporated

global isTimeInvariant

warning off
```

```
% Initilizing Values
globals
dU0 = 0;
V0 = 0;
W0 = 0;
P0 = 0;
Q0 = 0;
R0 = 0;
pAng0 = 0;
qAng0 = 0.1;
rAng0 = 0;

Ts = 1e7;
U0 = 1000;
for Tc = 1e6:1e6:2e8,
        initiateSysEqn

        % ------------------------------------------------------------------

        Tc,
        % Force Calculation
        F_a = zeros(1+round(L0/dl), ExpressionVectorSize);
        F_p = zeros(1+round(L0/dl), ExpressionVectorSize);
        F_y = zeros(1+round(L0/dl), ExpressionVectorSize);
        M_a = zeros(1+round(L0/dl), ExpressionVectorSize);
        M_p = zeros(1+round(L0/dl), ExpressionVectorSize);
        M_y = zeros(1+round(L0/dl), ExpressionVectorSize);
        DC_p = zeros(1+round(L0/dl), ExpressionVectorSize);

        FMthrust;            % Due to Thrust
        FMinertia;           % Due to engine inertia
        FMaerodynamic;       % Aerodynamic force
        % FMgravity
        % etc etc

        % ------------------------------------------------------------------


        % Add structural equations due to flexibility
        structuralEqn_p;     % On pitch plane

        % Equations for from net Force and Moment balance
        OverallForceMomentEqn;

        % ------------------------------------------------------------------
        eigArray(U0/100, :) = (eig(inv(AA)*BB))';

end

figure;
plot(eigArray, 'b');
title('Locus of poles of Open loop system with variation of Tc from 10^6 to 2*10^8');
```

Appendix – II

Mathematica code for numerically solving the Orr-Sommerfeld equation using Galerkin's Method

```
u[y_] := If[0 ≤ y ≤ 1, -0.5 y^3 + 1.5 y, If[y > 1, 1, 0]]

L[ϕ_] := (u[y] - c) (D[ϕ, {y, 2}] - α^2 ϕ) -
  ϕ D[u[y], {y, 2}] + (i/(α R)) (D[ϕ, {y, 4}] - 2 α^2 D[ϕ, {y, 2}] + α^4 ϕ)

φNo = 4

φ1[y_] := y^2 E^(-.1 y)

φ2[y_] := y^2.5 E^(-.08 y)

φ3[y_] := y^3 E^(-.06 y)

φ4[y_] := y^3.5 E^(-.04 y)

φ5[y_] := y^4 E^(-.02 y)

φ6[y_] := y^7 E^(-.001 y)

φ7[y_] := y^8 E^(-.0001 y)

φ8[y_] := y^9 E^(-.0001 y)

φ9[y_] := y^10 E^(-.00001 y)

φ10[y_] := y^11 E^(-.00001 y)

e[y_] := L[Sum[ℰi φi[y], {i, 1, φNo}]]

For[i = 1, i ≤ φNo, i++, pi = Integrate[e[y] φi[y], {y, 0, ∞}]]

m = Table[Coefficient[pi, ℰj+.0], {i, φNo}, {j, φNo}]

detM=Det[m]

ReplaceAll[detM,{R→1,α→1}]

Solve[ReplaceAll[detM,{R→1000,α→.4}]==0,c]

ContourPlot[Im[ReplaceAll[c,Solve[ReplaceAll[detM,{R→x,α→y}]==
0,c][[1]]]],{x,.1,1000},{y,.001,1}];
```



Created by *Mathematica* (May 5, 2006)

Appendix – III

Matlab code for numerically solving the Orr-Sommerfeld equation using 'Automated search of eigenvalues'

Following M-files were implemented in Matlab for finding the $c_i = 0$ contour in the α-R plane. The function 'zero_contour_search' is the entry point to the code.

globals.m:

```
function globals

global dy dyNos staGap

global phi0     % First Integration Pass Solution
global phi0_p phi0_pp phi0_ppp    % Values from First Integration Pass stored at
stations
global phi0_wall phi0_p_wall phi0_pp_wall phi0_ppp_wall  % Values from First
Integration Pass stored at the wall
global A4_0     % Initial A4 value chosen for First Integration Pass

global phiW     % Second Integration Pass Solution
global phiW_wall phiW_p_wall phiW_pp_wall phiW_ppp_wall  % Values from First
Integration Pass stored at the wall
global A2_W     % Initial A2 value chosen for Second Integration Pass

global tanh1

dy = .001;      % dy for Runge-Kutta method and U, U_pp
dyNos = 1/dy;   % Number of dy s over which the integration is done
staGap = 20;    % Number of dy gaps between two stations
staNos = dyNos/staGap;  % Number of Stations

A4_0 = 1e9;%1e32;
A2_W = 1;%1e-18;

tanh1 = tanh(1.0);
```

U.m:

```
function [U] = U(y)

global dyNos tanh1

if y > 1
    U = 1;
else
    discreetY = y;
    U = tanh(discreetY)/tanh1;
End
```

U_pp.m:

```
function [U_pp] = U_pp(y)

global dyNos tanh1

if y > 1
    U_pp = 0;
else
    %discreetY = fix(dyNos * y) / dyNos;
    discreetY = y;
    sechY = sech(discreetY);
    U_pp = -2 * (sechY.* sechY.* tanh(discreetY)) / tanh1;
end
```

zero_contour_search.m:

```
function zero_contour_search

format compact
sprintf('*****************************************\n'),

Rmin = 1;
Rmax = 8000;
alphaMin = .01;
alphaMax = 2.5;

ptNo = 1;
while ptNo <= 20,
    sprintf('New search point! \nPoint no: %d\n',ptNo),

    lambda = 1;
    dR = 10;
    dalpha = .01;

    R = Rmin + (Rmax-Rmin)*rand;
    alpha = alphaMin + (alphaMax-alphaMin)*rand;

    isConverge = 1;
    lastg = 1;
    iterNo = 0;
    while abs(lastg) > 1e-5,
        g = lambda * imag(search_c(R, alpha));

        g_R = lambda * imag(search_c(R+dR, alpha));
        g_alpha = lambda * imag(search_c(R, alpha+dalpha));

        dg_dR = (g_R - g)/dR;
        dg_dalpha = (g_alpha - g)/dalpha;

        if abs(dg_dR) < 1e-3 & abs(dg_dalpha) < 1e-3      %Flat g-surface
            lambda = 2/sqrt(abs(dg_dR*dg_dalpha));                  % Adaptive lambda
            sprintf('WARNING: Problem in convergence (Flat g-surface).... using
        adaptive lambda! lambda = %d', lambda),
        elseif abs(dg_dR) > 100 | abs(dg_dalpha) > 100    %Steep g-surface
            lambda = sqrt(abs(dg_dR*dg_dalpha))/2;                  % Adaptive lambda
            sprintf('WARNING: Problem in convergence (Steep g-surface).... using
        adaptive lambda! lambda = %d', lambda),
        else
            newR = R - g/dg_dR;
            newalpha = alpha - g/dg_dalpha;
            if g*lastg < 0
                R = (R + newR)/2;
                alpha = (alpha + newalpha)/2;
            elseif abs(g) < abs(lastg)
                R = newR;
                alpha = newalpha;
            else
                R = (R + newR)/2;
                alpha = (alpha + newalpha)/2;
                sprintf('WARNING: Suspecting local extreama'),
            end
            lambda = 1;
        end

        lastg = g;

        iterNo = iterNo + 1;
        if iterNo > 50
            sprintf('ERROR: No convergence could be achieved in 50 iterations....
        attempting once again afresh!\n'),
            isConverge = 0;
            break
        end
        sprintf('iterNo = %d  ,  g = %d', iterNo, g),
    end

    if isConverge == 1
        Rs(ptNo) = R;
        alphas(ptNo) = alpha;
        ptNo = ptNo + 1;
        sprintf('** Convergence achieved in %d iterations
        \n*************************************\n', iterNo),
    end
end

save 'C:\Documents and Settings\Subhrajit.SELF-37D14AD783\My Documents\B.Tech
        Project\Matlab\Solution\zero_contour_randomSearch_20pts_high_precision.mat' Rs
        alphas;

axis([0 log10(8000) 0 2.5]);
```

```
plot(log10(Rs), alphas,'.');
```

search_c.m:

```
function [search_c] = search_c(R, alpha)
% searches for c such that the boundary condition at y=0 is satisfied

global phi0_wall phi0_p_wall phi0_pp_wall phi0_ppp_wall  % Values from First
       Integration Pass stored at the wall
global phiW_wall phiW_p_wall phiW_pp_wall phiW_ppp_wall  % Values from First
       Integration Pass stored at the wall
global dy phi0 phiW

lambda= 0.5;
c = 0 + 0i;    %Starting value for c
delC = .1+.1i;

loopcount = 0;
while abs(delC) >= 1e-3 | abs(err) >= 1e-8,

    solution_I(c, R, alpha)
    solution_II(c, R, alpha)
    err = phi0_wall.*phiW_p_wall - phi0_p_wall.*phiW_wall;

    solution_I(c+delC, R, alpha)
    solution_II(c+delC, R, alpha)
    nxterr = phi0_wall.*phiW_p_wall - phi0_p_wall.*phiW_wall;

    if abs(err) < 1e-10
        break
    end

    if nxterr ~= err
        delC = -lambda * delC / (nxterr/err - 1);
    else
        delC = delC * 4;
    end
    c = c + delC;
    delC = .25*delC;

    loopcount = loopcount + 1;
    if loopcount > 100
        break
    end
end

search_c = c;

yy = dy:dy:1;
plot(yy, real(phi0 - phiW*phi0_wall/phiW_wall));
```

solution_I.m:

```
function [solution_I] = solution_I(c, R, alpha)
% Performs the first Integration Pass

global phi0     % First Integration Pass Solution
global phi0_p phi0_pp phi0_ppp    % Values from First Integration Pass stored at
       stations
global phi0_wall phi0_p_wall phi0_pp_wall phi0_ppp_wall  % Values from First
       Integration Pass stored at the wall
global A4_0     % Initial A4 value chosen for First Integration Pass
global dy dyNos staGap


p4 = -alpha .* sqrt(1 + i*R.*(1-c)./alpha);
p4_expp4 = p4 .* exp(p4);
p4sq = p4 .* p4;

phi(1) = A4_0 * exp(p4);
phi(2) = A4_0 * p4_expp4;
phi(3) = A4_0 * p4 * p4_expp4;
phi(4) = A4_0 * p4sq * p4_expp4;

stationCount = 0;
for stepNo = dyNos:-1:1,
    phi0(stepNo) = phi(1);
    if mod(stepNo, staGap) == 0     % A station has been reached!.... Storing values
        stationCount = stationCount + 1;
        phi0_p(stationCount) = phi(2);
        phi0_pp(stationCount) = phi(3);
        phi0_ppp(stationCount) = phi(4);
    end
```

```
    phi = getNext_phi(phi, (stepNo-1)*dy, c, R, alpha);
end

phi0_wall = phi(1);
phi0_p_wall = phi(2);
phi0_pp_wall = phi(3);
phi0_ppp_wall = phi(4);
```

solution_II.m

```
function [solution_II] = solution_II(c, R, alpha)
% Performs the second Integration Pass

global phiW     % Second Integration Pass Solution
global A2_W     % Initial A2 value chosen for Second Integration Pass
global phiW_wall phiW_p_wall phiW_pp_wall phiW_ppp_wall  % Values from First
       Integration Pass stored at the wall
global dy dyNos staGap


p2 = -alpha;
p2_expp2 = p2 .* exp(p2);
p2sq = p2 .* p2;

phi(1) = A2_W * exp(p2);
phi(2) = A2_W * p2_expp2;
phi(3) = A2_W * p2 * p2_expp2;
phi(4) = A2_W * p2sq * p2_expp2;

for stepNo = dyNos:-1:1,
    phiW(stepNo) = phi(1);
    phi = getNext_phi(phi, (stepNo-1)*dy, c, R, alpha);
end

phiW_wall = phi(1);
phiW_p_wall = phi(2);
phiW_pp_wall = phi(3);
phiW_ppp_wall = phi(4);
```

getNext_phi.m

```
function [getNext_phi] = getNext_phi(this_phi, y, c, R, alpha)
% Returns the value of phi and it's derivetives in the next step of
% Runge-Kutta iteration (i.e. at the previous value of y).
% Here the 4 values are [phi phi_p phi_pp phi_ppp]
global dy

alphasq = alpha.*alpha;

getNext_phi(1) = this_phi(1) - dy*this_phi(2);
getNext_phi(2) = this_phi(2) - dy*this_phi(3);
getNext_phi(3) = this_phi(3) - dy*this_phi(4);
getNext_phi(4) = this_phi(4) - dy * (i*R.*alpha.*((U(y)-c).*(this_phi(3)-
       alphasq.*this_phi(1)) - U_pp(y).*this_phi(1)) + 2*alphasq.*this_phi(3) -
       alphasq.*alphasq.*this_phi(1));
```

References

[1] A L Greensite, *Analysis and Design of Space Vehicle Flight Control*, Macmillan and Co. Ltd., 1970.

[2] A Tsourdos, B A White, *Adaptive flight control design for nonlinear missile*, Control Engineering Practice 13, 373–382, 2005.

[3] S H Pourtakdoust, N Assadian, *Investigation of thrust effect on the vibrational characteristics of flexible guided missiles*, Journal of Sound and Vibration 272, 287–299, 2004.

[4] T P Chang, *Dynamic response of space structures under random excitation*, Computers & Structures Vol. 48, No. 4, pp. 575-582. 1993

[5] A Guran, K Ossia, *On the stability of a flexible missile under an end thrust*, Math Comput. Modelling, Vol. 14, pp. 965-968, 1990.

[6] Dr. Hermann Schlichting, *Boundary Layer Theory*, McGraw Hill, 1968.

[7] Robert Betchov, William O. Criminale, Jr., *Stability of Parallel Flows*, Academic Press, 1967.

[8] T B Benjamin, *Effect of a flexible boundary on hydrodynamic stability*, J. Fluid Mechanics, 9, 4, 513-532, 1960.

[9] A F Schmitt, et al., *Approximate Transfer function for Flexible Booster and Autopilot Analysis*, WADD TR-61-93, April 1961.

[10] R Betchov, A B Szewczyk, *Stability of a shear layer between parallel streams*, Physics of Fluids 6, 1391-1396,1963.

[11] R P Nachtsheim, *An initial value method for numerical treatment of the Orr-Sommerfeld equation for the case of plane Poiseuille flow*, N.A.S.A.Tech. Note D-2414.

[12] C T Leondes, *Guidance and Control of Aerospace Vehicles*, New York, McGraw-Hill Book Company, 1963.