Simple Vibration Problems with MATLAB (and Some Help from MAPLE)

Original Version by Stephen Kuchnicki

December 7, 2009

Contents

Preface

This document is companion to the text: Mechanical Vibration: Analysis, Uncertainties and Control, by Haym Benaroya and Mark Nagurka, CRC Press 2010; contact: benaroya@rci.rutgers.edu; mark.nagurka@marquette.edu

Chapter 1

Introduction

This document is intended as a companion to *Mechanical Vibration: Analysis, Uncertainties, and Control* by Haym Benaroya and Mark Nagurka. This companion draws heavily upon the Matlab software package, produced by the MathWorks, Inc. We do not intend to teach Matlab in this work; rather, we wish to show how a vizualization tool like Matlab can be used to aid in solution of vibration problems, and hopefully to provide both the novice and the experienced Matlab programmer a few new tricks with which to attack their problems of interest.

Matlab (*Mat*rix *Lab*oratory) was born from the LINPACK routines written for use with C and Fortran. The Matlab package provides both command-line and programming language interfaces, allowing the user to test simple statements at the command prompt, or to run complicated codes by calling a function name. Matlab is excellent for handling matrix quantities because it assumes every variable is an array. Thus, calling on the multiplication operator alone causes Matlab to attempt matrix, not scalar multiplication. Also, Matlab includes special "array operators" that allow multiplication of arrays on an element-by-element basis. For example, if we set the variable $a = [1\ 2\ 3]$ and $b = [4\ 5\ 6]$, we can perform the matrix multiplications:

$$c = a * b' \tag{1.1}$$

$$d = a' * b \tag{1.2}$$

(Note that the apostrophe is the transpose operator in Matlab.) The result c would be a scalar (specifically, 32). The variable d would contain a 3-by-3 matrix, whose rows would be scalar multiples of b. However, what if the values stored in a are three masses, and those in b their corresponding accelerations? If we wished to find the force on each mass, we would need to multiply the first element of a by the first element of b, and so on for the second and third elements. Matlab provides "array multiplication" for such an instance:

$$e = a. * b \tag{1.3}$$

Take special note of the period between a and the asterisk. This tells Matlab to ignore its usual matrix multiplication rules, and instead create e by multiplying the corresponding elements of a and b. The result here would be $e = [4$ $10\ 18]$. This is one of the more useful specialized commands in Matlab, and one we will use frequently. Other commands will be discussed as they arise.

The vast majority of these examples will use Matlab in its programming mode. The general format is to introduce a problem, with reference to the text where applicable, and to show the analytic solution (if derivable). Several figures follow each example, showing results of each vibration problem under different sets of parameters and making use of Matlab's integrated graphics capabilities. Finally, the Matlab code used to generate the figures is presented, with comments explaining what was done, why it was done, and other ways it could have been done in Matlab. The code should look somewhat familiar to those who have used C and Fortran in the past; Matlab's language shares several common structures with both of these languages, making it relatively easy for an experienced C or Fortran programmer to learn Matlab.

One distinction to make with Matlab programming is between *script* m-files and *function* m-files. Both are sets of commands that are saved in a file. The differences are that variables used in a script file are retained in the Matlab workspace and can be called upon after the script completes. Variables within a function m-file cannot be used outside of the function unless they are returned (much like C functions or Fortran subroutines). Additionally, a function must begin with the line `function` $output = function\_name(var1, var2, ...varN)$. This line tells the Matlab interpreter that this file is a function separate from the workspace. (Again, the form should look familiar to Fortran and C programmers.)

Chapter 2

SDOF Undamped Oscillation

The simplest form of vibration that we can study is the single degree of freedom system without damping or external forcing. A sample of such a system is shown in Figure 2.1. A free-body analysis of this system in the framework of Newton's second law, as performed in Chapter 2 of the textbook, results in the following equation of motion:

$$m\ddot{x} + kx = 0. \tag{2.1}$$

(In general, we would have the forcing function $F(t)$ on the right-hand side; it's assumed zero for this analysis.) Dividing through by m, and introducing the parameter $\omega_n = \sqrt{k/m}$, we obtain a solution of the form

$$x(t) = A\sin(\omega_n t + \phi), \tag{2.2}$$

or, in terms of the physical parameters of the system, we have

$$x(t) = \frac{\sqrt{\omega_n^2 x_o^2 + \dot{x}_o^2}}{\omega_n}\cos(\omega_n t - \tan^{-1}\frac{\dot{x}_o}{\omega_n x_o}). \tag{2.3}$$

From this, we see that the complete response of an undamped, unforced, one degree of freedom oscillator depends on three physical parameters: ω_n, x_o, and $\dot{x}_o$: the natural frequency, initial velocity, and initial displacement, respectively.

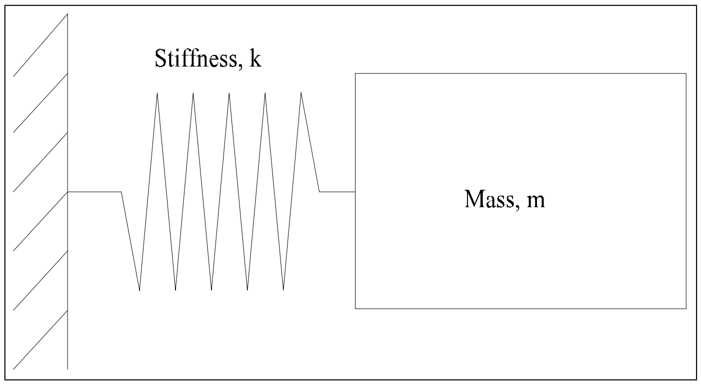

Figure 2.1: Typical single degree of freedom free oscillator.

From the definition of the natural frequency, we see that it is inversely proportional to $\sqrt{m}$, and is directly proportional to $\sqrt{k}$. Variation of mass or stiffness, then, will cause a variation in the frequency of vibration. The graphs on the following pages demonstrate these sensitivities. Figure 2.2 shows the variation of the vibrational characteristics for an increasing mass while stiffness remains constant. Figure 2.3 shows how the oscillatory behavior changes for increasing stiffness with mass constant. For both Figures 2.2 and 2.3, the initial velocity used was $\dot{x}_o = 1$, and the initial displacement $x_0 = 3$. Note that the frequency decreases with increasing mass, but increases with increasing stiffness, as expected.

Looking at Equation 2.3, it is evident that the phase angle and maximum amplitude are also functions of the natural frequency. Again, referring to Figure 2.2, the maximum amplitude decreases with increasing mass, due to the corresponding reduction in natural frequency. As a result, the phase shift diminishes, with the peak of oscillation becoming nearer to $t = 0$. Note that the maximum displacement would occur at $t = 0$ if the initial velocity were zero. It is easily verified that, for this case, the parameter A (from Equation 2.2, above) reduces to x_o, and the phase angle becomes $tan^{-1}(0/x_o^2)$, or $0°$.

In Figure 2.3, the maximum amplitude increases with increasing stiffness, due to the increase in natural frequency. The phase angle also increases with the stiffness, so the maximum amplitude of oscillation occurs at an increasingly later time with increasing stiffness.

The MATLAB code used to produce the graphs follows. The `input` statements in the first few lines demonstrate one type of interactive programming available in MATLAB. The portion of the statement inside the quotation marks will appear on the screen when the program is run, so user prompts are easily added and explained. The initialization and use of the mass matrix is demonstrated in two ways. If the variable $matflag$ is set to zero, then the masses are each given scalar values. If this flag is set to one, then the masses are initialized as part of an array. This is done to demonstrate the use of MATLAB array variables and to show how they can help streamline your code. Note also that the `if..then` structure in MATLAB is similar to that of Fortran in that these statements must be closed with an `end` statement, as must any other loop (`for` or `do`, for example). We also demonstrate MATLAB's plotting routines via the `subplot` command, allowing plots for all three masses to be placed on the same axes. The second figure below was produced by modifying the code below to take three stiffnesses and one mass. The modifications necessary are left as an exercise for the reader.

Finally, those unfamiliar with MATLAB are probably wondering what all the semicolons are after each line. By default, MATLAB prints the results of each operation to the screen. However, placing a semicolon at the end of a line suppresses this output. Since printing to screen takes time and memory for MATLAB to perform, suppressing screen output of intermediate results increases computational speed. (The reader is invited to remove all the semicolons from the program to see the difference!) This feature is also a very useful debugging tool. Instead of inserting and removing print statements to check intermediate calculations, the programmer can insert and delete semicolons.

Program 1-1: varym.m

```
% The first few lines obtain the desired parameter values.
% This could also be done by using assignment statements in
% the program, but then the program would need to be
% edited to change the parameters. As written, parameters
% can be changed by re-running the program.
%
matflag=0; % Set to 1 to activate mass array, zero for mass scalars.
k=input('Enter the stiffness value. ');
x0=input('Enter the initial displacement. ');
v0=input('Enter the initial velocity. ');
tf=input('Enter the time duration to test, in seconds. ');
if (matflag)
    for i=1:3
        m(i)=input(['Enter mass value ', num2str(i),'. ']);
        % More about the 'num2str' command below.
    end
else
    m1=input('Enter the first mass value. ');
    m2=input('Enter the second mass value. ');
    m3=input('Enter the third mass value. ');
end
%
% This loop initializes the natural frequency values. This
% is more streamline by making a mass matrix [m(1) instead
% of m1, etc.], as shown. Note that the natural frequency is stored
% in a matrix.
%
if (matflag)
    wn=sqrt(k./m);
%
% Array division, akin to the array
% multiplication described above. This one line produces a three-
% element array of natural frequencies. Compare to
% the machinations below.
%
else
    for i=1:3
        switch i % Analogous to the C command; saves a lot of "if" state-
ments.
            case 1
                m=m1;
            case 2
                m=m2;
            case 3
                m=m3;
            end
        wn(i)=sqrt(k/m);
        end
end
%
% Now, the values for A and phi in the expression
% x(t)=Asin(wn*t + phi) are evaluated. Notice that, in
% order for the cosine function to evaluate properly in
% MATLAB, the wn vector is used element by element
% (thus, the loop).
%
t=0:tf/1000:tf; % We need only initialize the time increment once.
for j=1:3
        a=sqrt(wn(j)^2*x0^2+v0^2)/wn(j); % The caret is the power operator.
        phi=atan2(v0,wn(j)*x0);        % atan2 is the four-quadrant arctangent.

        x(j,:)=a*cos(wn(j)*t-phi);
end
%
% Since this program was made to compare different
% parameters, a subplot format makes sense. If the number
% of varied masses is changed, the subplot statement must
% also be.
%
subplot(3,1,1)
plot(t,x(1,:))
%
% This line demonstrates the use of the num2str command.
% This command allows the value of a variable to be used
% in a title or other text. Note that the command doesn't
% work unless the text intended as a title is enclosed in
% brackets as well as parentheses.
%
if (matflag)
        title(['Response for m=',num2str(m(1)), ', k=', num2str(k)])
else
        title(['Response for m=',num2str(m1), ', k=', num2str(k)])
end
ylabel('Response x')
grid
subplot(3,1,2)
plot(t,x(2,:))
if (matflag)
        title(['Response for m=',num2str(m(2)), ', k=', num2str(k)])
else
        title(['Response for m=',num2str(m2), ', k=', num2str(k)])
```

```
end
ylabel('Response x')
grid subplot(3,1,3)
plot(t,x(3,:))
if (matflag)
    title(['Response for m=',num2str(m(3)), ', k=', num2str(k)])
else
    title(['Response for m=',num2str(m3), ', k=', num2str(k)])
end
ylabel('Response x')
xlabel('Time, seconds')
grid
```

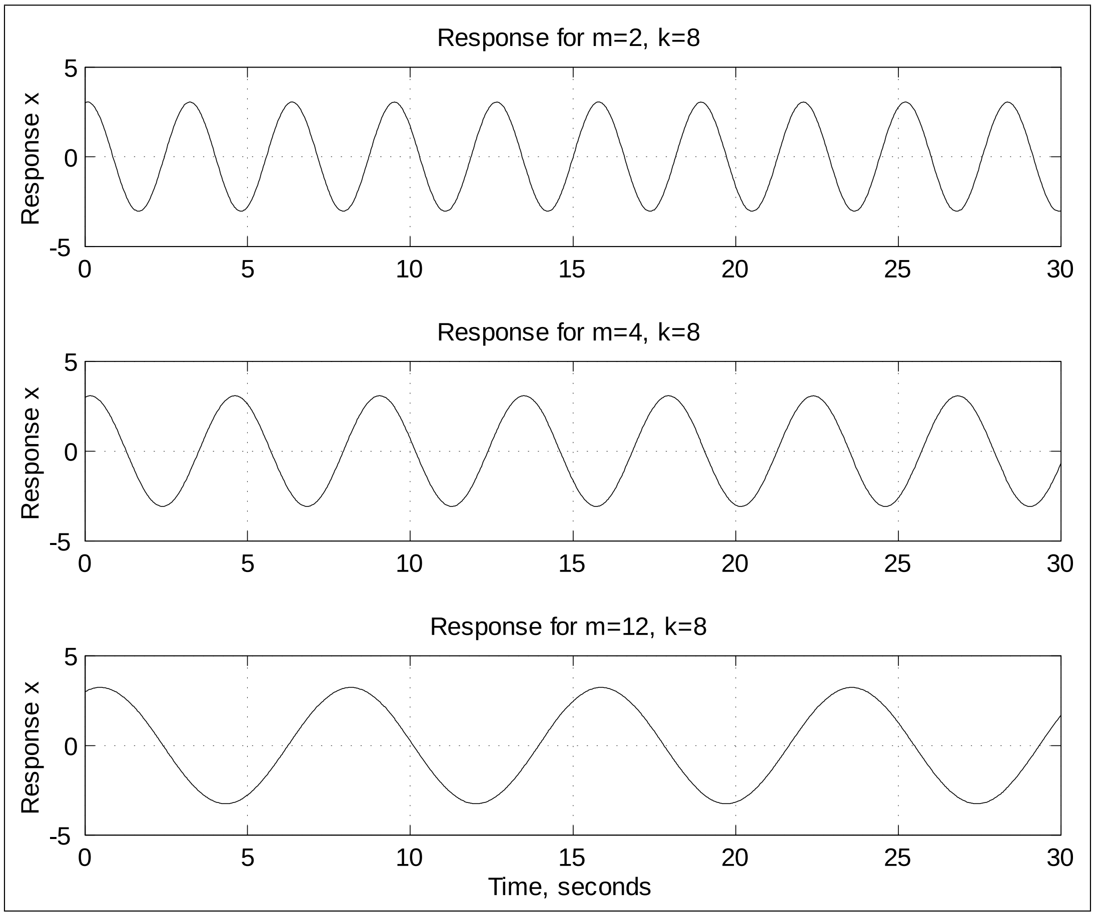

Figure 2.2: Responses for three different masses.

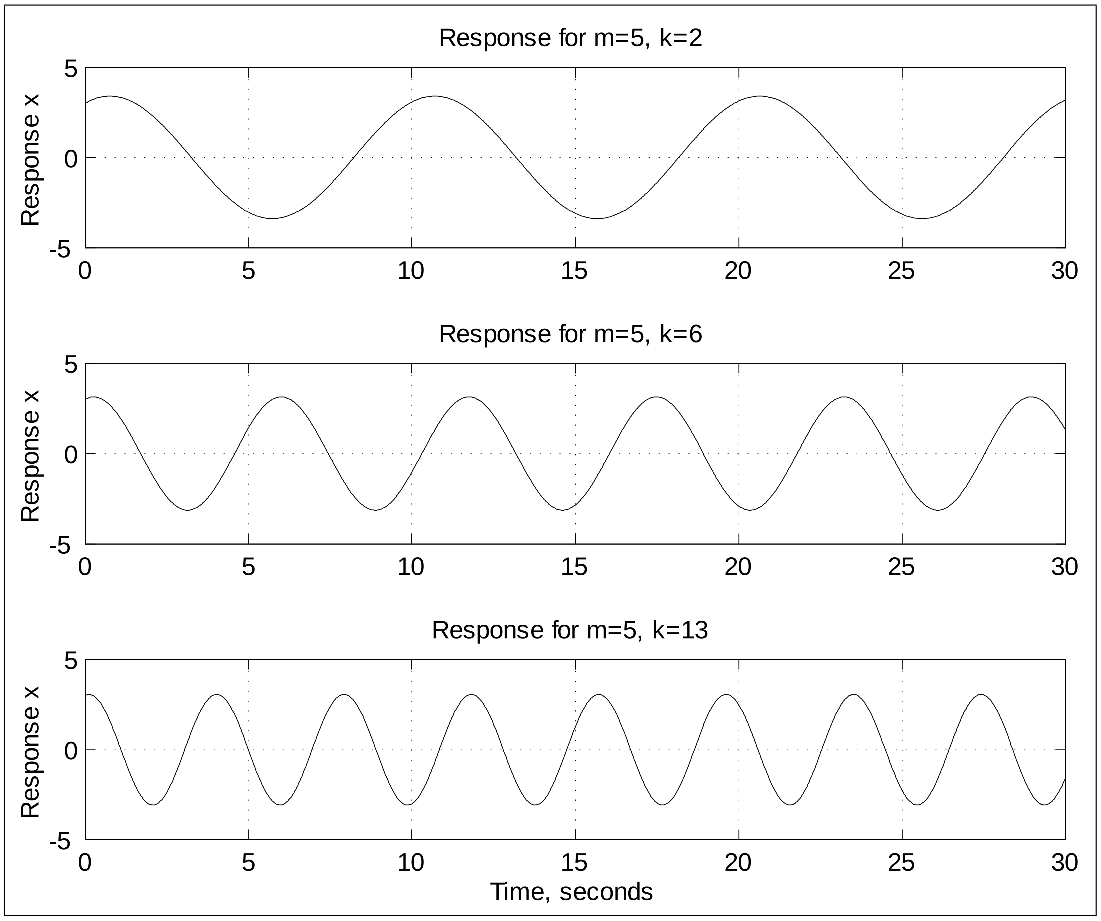

Figure 2.3: Responses for three different stiffnesses.

Chapter 3

A Damped SDOF System

In the previous example, we examined the response of an undamped single degree of freedom oscillator subject to varied mass and stiffness parameters. Here we will do the same for a damped single degree of freedom system.

Again, we begin with the equation of motion, given the system of Figure 3.1. A simple static analysis finds that our equation is:

$$m\ddot{x} + c\dot{x} + k\ x = 0. \tag{3.1}$$

If we divide through by m, we introduce the dimensionless parameters ω and ζ:

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2\ x = 0. \tag{3.2}$$

In the above, ω_n represents the undamped natural frequency, and ζ is the viscous damping ratio. For the purposes of this example, we will assume the underdamped case ($\zeta < 1$). The solution to this equation is:

$$x(t) = Ae^{-\zeta\omega_n t}\sin(\omega_d t + \phi). \tag{3.3}$$

In Equation 3.3, ω_d is the damped natural frequency, equal to $\omega_n\sqrt{1-\zeta^2}$.

This equation is more useful if we write all of the terms as functions of parameters ω_n and ζ:

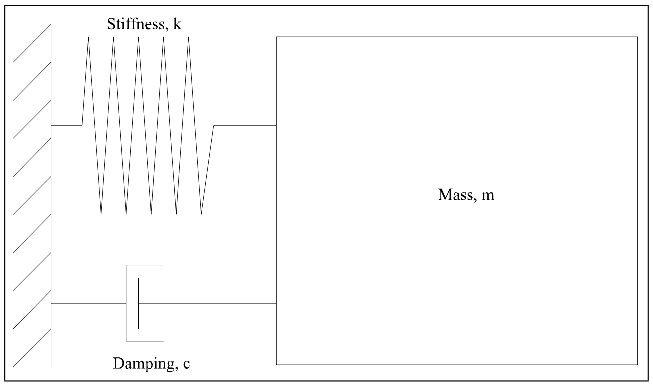

Figure 3.1: Typical damped single degree of freedom oscillator.

$$\begin{aligned} x(t) &= \sqrt{\frac{(v_o + \zeta\omega_n x_o)^2 + \left(x_o\omega_n\sqrt{1-\zeta^2}\right)^2}{\left(\omega_n\sqrt{1-\zeta^2}\right)^2}}\, e^{(-\zeta\omega_n t)} \\ &\quad \times \sin\left[\left(\omega_n\sqrt{1-\zeta^2}\right)t + \tan^{-1}\left(\frac{x_o\omega_n\sqrt{1-\zeta^2}}{v_o+\zeta\omega_n x_o}\right)\right]. \end{aligned} \tag{3.4}$$

In this expression, the term $\omega_n\sqrt{1-\zeta^2}$ has been substituted for ω_d.

While this equation admittedly looks intimidating, note that it only depends on four quantities: x_o, v_o, ω_n, and ζ. Note the similarity between the parameters identified here and the ones relevant to the undamped case; the only difference is the addition of the viscous damping coefficient. So, we have already seen the effects of changing all of the parameters except the damping coefficient. Figure 3.2 gives the variation of the response with increasing viscous damping coefficient for $x_o = 3, v_o = 1$, and $\omega_n = 7$. Note how quickly the response becomes virtually zero; this occurs within ten seconds, even for a damping coefficient as small as 0.05!

The MATLAB code used to generate the figure follows. This code is, in many ways, similar to the code used earlier for the mass and stiffness parameter studies. However, since both the exponential term and the sine term in our solution (Equation 3.3) depend on time, they are both vector quantities. To multiply them properly, we must use the array multiplication operator, '.*'. The code again makes use of `input` statements to initialize the parameters. This time, however, the damping ratios are entered in array form only. The algorithm chosen to produce the plots works only for the underdamped case; since we have the term $\omega_d = \left(\omega_n\sqrt{1-\zeta^2}\right)$ in the denominator of our response equation, $\zeta = 1$ will cause division by zero, and $\zeta > 1$ will give an imaginary damped natural frequency. We employ some defensive programming to ensure allowable values for the damping ratio are entered. Conveniently, this gives us a reason to introduce the `while` loop, analogous to the "do while" loop in Fortran. The code checks each value `zeta(zi)` to see if it lies between zero and one. If not, a somewhat abrupt reminder is sent to the screen, and the loop asks for a new entry. Otherwise, it continues to the next damping ratio value. The logical and relational operators in MATLAB are for the most part intuitive. Thus, `if a>0` simply means "if a is greater than zero." A full list can be found by typing `help ops` at the MATLAB prompt; we will explain the ones we use in comments.

Program 2-1: varyzeta.m

```
%Initial value entry.
%The while loop in the zeta initialization section prevents
%certain values for zeta from being entered, since such
%values would crash the program.
%
wn=input('Enter the natural frequency. ');
x0=input('Enter the initial displacement. ');
v0=input('Enter the initial velocity. ');
tf=input('Enter the time duration to test, in seconds. ');
for zi=1:3
    zeta(zi)=12;
    while(zeta(zi)<0 | zeta(zi)>=1) % The pipe (|) means "or".
        zeta(zi)=input(['Enter damping coefficient value ', num2str(zi),'.
']);
        if (zeta(zi)>=1 | zeta(zi)<0)
            fprintf('Zeta must be between 0 and 1!');
            zeta(zi)=12;
        end
    end
end
%
%Now, having ωn and ζ, the ωd values can be found.
%
for i=1:3
    wd(i)=wn*sqrt(1-zeta(i)^2);
end
%
%Solving for the response. Note the use of the array
%multiplication command (.*) in the expression for x(t).
%This command is necessary, else the program gives a
%multiplication error.
%
t=0:tf/1000:tf;
for j=1:3
    a=sqrt((wn*x0*zeta(j)+v0)^2+(x0*wd(j))^2)/wd(j);
    phi=atan2(wd(j)*x0,v0+zeta(j)*wn*x0);
    x(j,:)=a*exp(-zeta(j)*wn*t).*sin(wd(j)*t+phi);
end
%
%Now, the program plots the results in a subplot format.
%
subplot(3,1,1)
plot(t,x(1,:))
title(['Response for zeta=',num2str(zeta(1))])
ylabel('Response x')
```

```
grid
subplot(3,1,2)
plot(t,x(2,:))
title(['Response for zeta=', num2str(zeta(2))])
ylabel('Response x')
grid
subplot(3,1,3)
plot(t,x(3,:))
title(['Response for zeta=', num2str(zeta(3))])
ylabel('Response x')
xlabel('Time, seconds')
grid
```

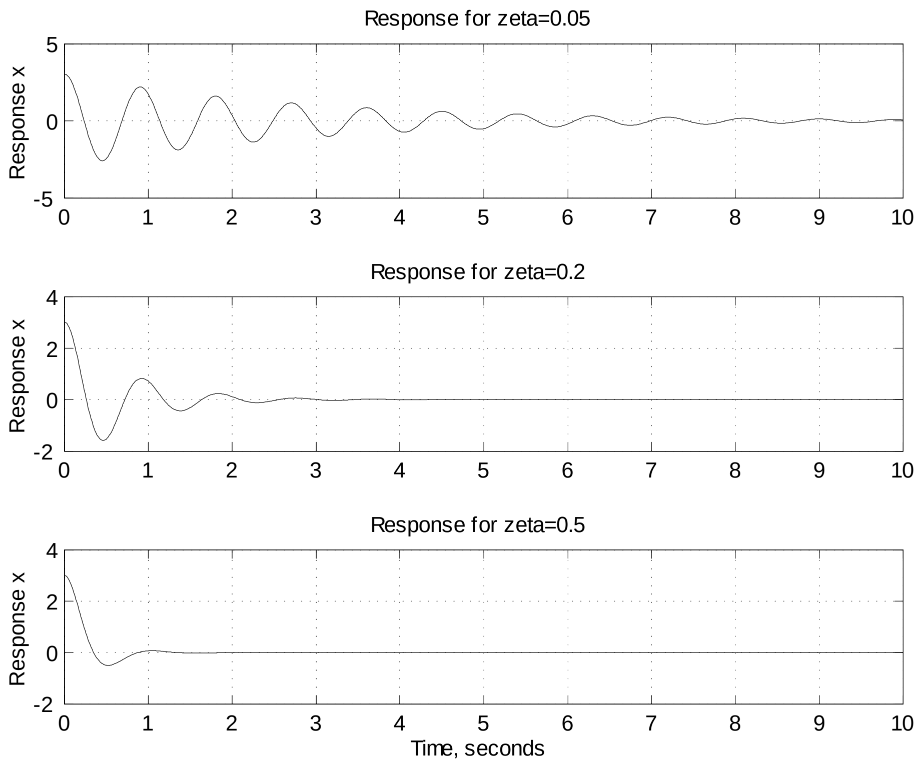

Figure 3.2: Responses for various zeta values.

Chapter 4

Overdamped SDOF Oscillation

The previous example is valid only for the underdamped case, $\zeta < 1$. But what happens when this condition is not met? There are two cases to explore: $\zeta > 1$ (the overdamped case), and $\zeta = 1$ (the critically damped case). First, we need to see how these cases arise in order to understand what the results will be. The equation of motion of a damped single degree of freedom oscillator is:

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = 0. \tag{4.1}$$

Assume a solution of the form $Ae^{\lambda t}$, substitute it appropriately into Equation 4.1, and obtain the quadratic formula defining possible values for λ:

$$\lambda = -\zeta\omega_n \pm \omega_n\sqrt{\zeta^2 - 1}. \tag{4.2}$$

Note that the quantity inside the radical is always greater than zero, since we have assumed $\zeta > 1$. Therefore, the solution of the equation of motion is

$$x(t) = e^{-\zeta\omega_n t}(a_1 e^{\omega_n t\sqrt{\zeta^2-1}} + a_2 e^{-\omega_n t\sqrt{\zeta^2-1}}). \tag{4.3}$$

If we again take our initial displacement as x_o and initial velocity v_o, the constants a_1, a_2 become

$$a_1 = \frac{-v_o + (-\zeta + \sqrt{\zeta^2 - 1})\omega_n x_o}{2\omega_n\sqrt{\zeta^2 - 1}},\ a_2 = \frac{v_o + (\zeta + \sqrt{\zeta^2 - 1})\omega_n x_o}{2\omega_n\sqrt{\zeta^2 - 1}}. \tag{4.4}$$

Equation 4.3 is a decaying exponential and the system will simply return to its initial position instead of oscillating about the equilibrium. This is shown in Figure 4.1. Note that if $\zeta = 1$, a singularity exists in the constants; a second independent solution must be found. From our knowledge of ordinary differential equations, we can find

$$x(t) = (a_1 + a_2 t)e^{-\omega_n t}, \tag{4.5}$$

where $a_1 = x_o$, and $a_2 = v_o + \omega_n x_o$.

Figure 4.1 was generated for $\omega_n = 7$, $x_o = 3$, and $\nu_o = 1$. Notice how the critically damped response returns to equilibrium faster than the others. For the plots in the figure, the motion with critical damping is stopped after about two seconds, while the others do not reach equilibrium until more than eight seconds. This is the distinguishing characteristic of the critically damped case. Note also that the motion of the masses is, as expected, purely exponential; there is no oscillation, only a decay of the response to equilibrium.

The MATLAB code is again presented below. The astute reader may ask why the codes for underdamped, critically damped, and overdamped vibration could not be combined into a single code. Certainly they can be; the modifications are left to the reader. The resulting code should reject negative values for the damping ratio. (Hint: You may find the `switch` command to be useful, especially when used with its `otherwise` case.)

Program 3-1: overdamp.m

```
%Initial value entry. The while loop in the zeta initialization
%section prevents illegal values for zeta from being entered,
%since such values would crash the program.
%
wn=input('Enter the natural frequency. ');
x0=input('Enter the initial displacement. ');
v0=input('Enter the initial velocity. ');
tf=input('Enter the time duration to test, in seconds. ');
for zi=1:3
    zeta(zi)=0.12;
    while(zeta(zi)<1)
        zeta(zi)=input('Enter a damping coefficient value. ');
        if zeta(zi)<1
            fprintf('Zeta must be greater than 1!');
            zeta(zi)=0.12;
        end
    end
end
%
%Solving for the response. Notice that the variable den is used
%for the denominator of the constants, as well as the exponent
%values. If a variable can be created that will serve several
%purposes, it will save time and typing. (imagine typing the
%expression for den three times and you'll understand).
%
t=0:tf/1000:tf;
for j=1:3
    if zeta(j)>1
        a1n=-v0+(-zeta(j)+(zeta(j)^2-1)^0.5)*wn*x0;
        a2n=v0+(zeta(j)+(zeta(j)^2-1)^0.5)*wn*x0;
        den=wn*(zeta(j)^2-1)^0.5;
        a1=a1n/(2*den);
        a2=a2n/(2*den);
        x(j,:)=(a1*exp(-den*t)+a2*exp(den*t)).*exp(-zeta(j)*wn*t);
    elseif zeta(j)==1
        a1=x0;
        a2=v0+wn*x0;
        x(j,:)=(a1+a2*t).*exp(-wn*t);
    end
end
%Now, the program plots the results in a subplot format.
%
subplot(3,1,1)
plot(t,x(1,:))
title(['Response for zeta=',num2str(zeta(1))])
```

```
ylabel('Response x')
grid
subplot(3,1,2)
plot(t,x(2,:))
title(['Response for zeta=', num2str(zeta(2))])
ylabel('Response x')
grid
subplot(3,1,3)
plot(t,x(3,:))
title(['Response for zeta=', num2str(zeta(3))])
ylabel('Response x')
xlabel('Time, seconds')
grid
```

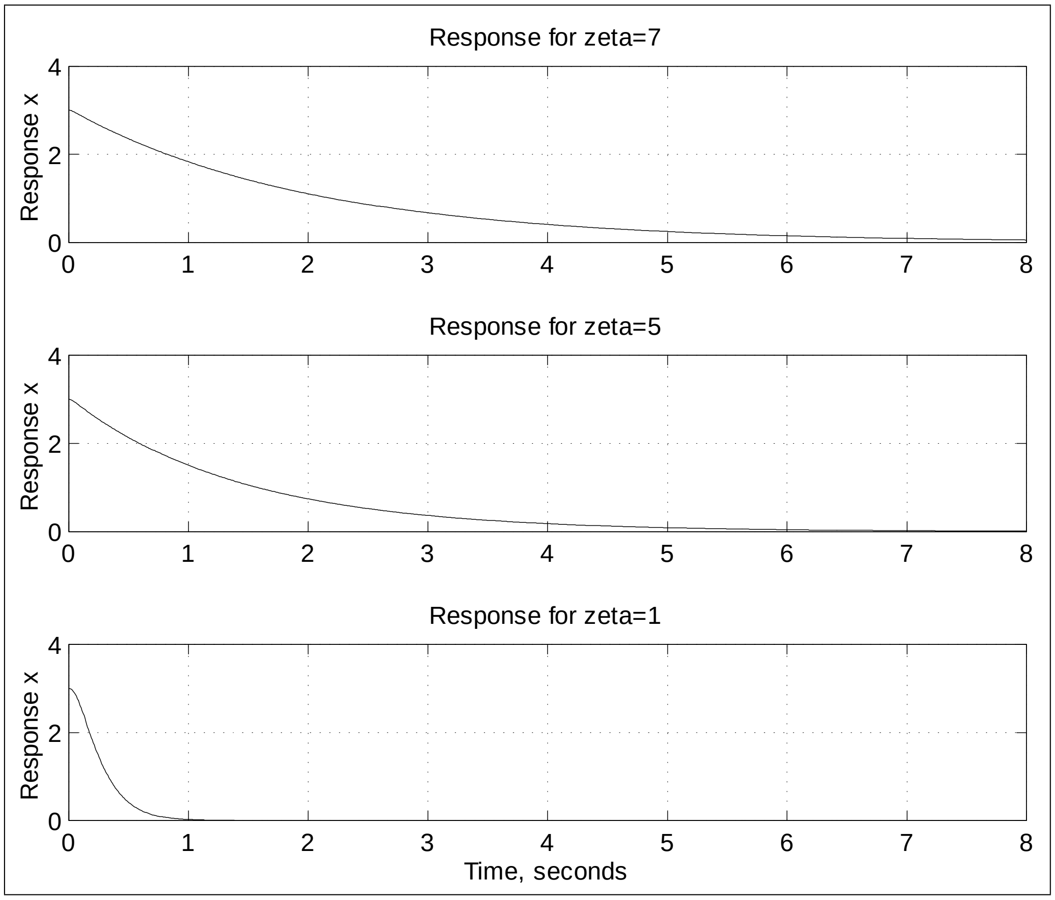

Figure 4.1: Response of three overdamped systems.

Chapter 5

Harmonic Excitation of Undamped SDOF Systems

In the previous examples, we examined the responses of single degree of freedom systems which were not subjected to an external force. Now, we will examine the effects of an external force on the system. We begin with the simplest form of external forcing: the harmonic load.

The system under consideration is shown in Figure 5.1. The forcing function is assumed to be of the form $F(t) = F_o \cos \omega t$, where ω is the driving frequency. For the case with no damping, Newton's Second Law gives us the equation of motion $m\ddot{x} + kx = F_o \cos \omega t$, or

$$\ddot{x} + \omega_n^2 x = f_o \cos \omega t, \tag{5.1}$$

where $f_o = F_o/m$. We know the solution for the response $x(t)$ to be:

$$x(t) = A_1 \sin \omega_n t + A_2 \cos \omega_n t + \frac{f_o}{\omega_n^2 - \omega^2} \cos \omega t, \tag{5.2}$$

where

$$A_1 = \frac{v_o}{\omega_n},\ A_2 = x_o - \frac{f_o}{\omega_n^2 - \omega^2}. \tag{5.3}$$

The key parameters which define the response are the natural and driving frequencies, or more precisely, their ratio $\omega/\omega_{n.}$ Figure 5.2 shows the effect of varying driving frequency ω for a given natural frequency, and Figure 5.3 does

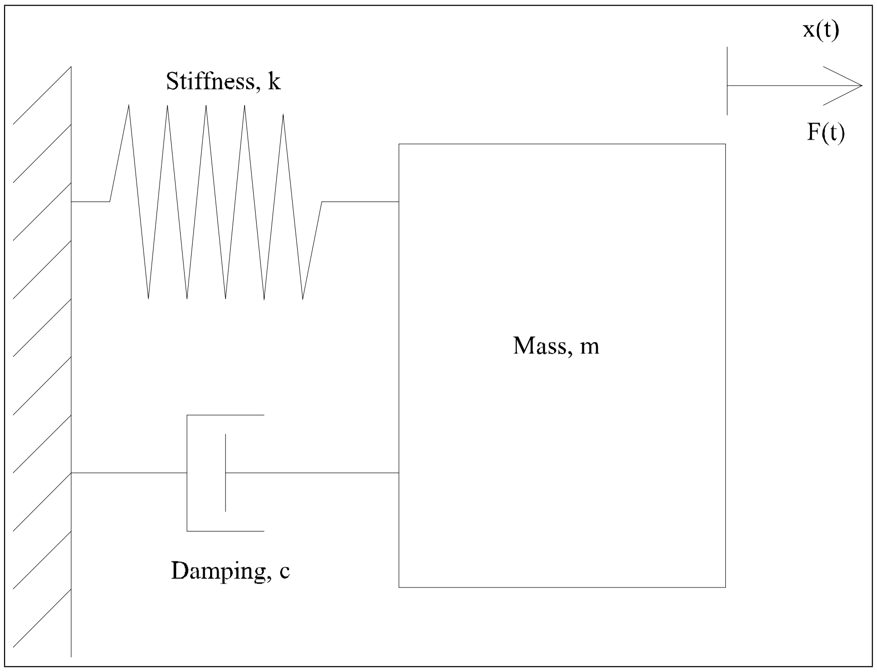

Figure 5.1: Model of a single degree of freedom subject to an external force.

the same for various natural frequencies ω_n. In these calculations, $v_o = 0, x_o = 0, F_o = 6$. Making the initial conditions zero allows us to better see the effects of varying frequencies.

Note that two of the three constants in the expression for $x(t)$ involve the difference between the frequencies. This gives rise to two interesting phenomena: *beats* and *resonance*. Beats occur when the natural frequency and the driving frequency are close but not equal. The result is then a rapid oscillation with a slowly varying amplitude, as shown in Figure 5.4. Note how the both the rapid oscillation and the slow change of the amplitude both vary along a sinusoid.

When the driving and natural frequencies are equal, $\omega = \omega_n$, resonance is the result. The third term in Equation 5.2 is not valid as a particular solution of the governing equation of motion. Instead, the particular solution is

$$x_p(t) = \frac{f_o}{2\omega_n} t \sin \omega_n t. \tag{5.4}$$

This term is the heart of the resonance problem; the amplitude of oscillation will increase without limit. In a real system, the stiffness element has a certain yield point which will be met and exceeded by a resonant vibration. Figure 5.5 shows resonant vibration.

The MATLAB code below uses the `subplot` command more elegantly than we had in earlier examples. Instead of typing out three separate sets of plotting commands, the subplots are done in a loop. A simple (and natural) modification to this program would be to change the number of plots in the subplot. This could be done by introducing an integer variable *nplot* and changing the loops to run from one to *nplot* instead of one to three. From a practical point of view, though, introducing too many plots to the figures would make the plot unreadable, so care must be taken. Also, we've changed the parameter values from dynamic inputs to static values. They can easily be changed back to user inputs.

Program 4-1 - varywdr.m

```
%This program is very straightforward. The program takes
%the necessary input values to solve Equation
%5.3 for three different driving
%frequencies, checks for resonance (since we don't want to
%handle that problem yet), and plots.
%
%This file varies driving frequency of a single degree
%of freedom undamped oscillator at a set natural
%frequency. The initial displacement is 3, the velocity is
%1, the force magnitude per unit mass is 6, and the
%natural frequency is 7.
wn=7;
x0=0;
v0=0;
f0=6;
tf=10;
wdr=zeros(3,1);
x=zeros(3,1001);
for i=1:3;
    wdr(i)=wn; % This is how we initialize our while loop.
    while wdr(i)==wn;
        wdr(i)=input('Enter the driving frequency. ');
        if wdr(i)==wn;
            fprintf('This will produce resonance!!')
        end
    end
end
t=0:tf/1000:tf;
for j=1:3
    A1=v0/wn;
    A2=x0-(f0/(wn^2-wdr(j)^2));
    A3=f0/(wn^2-wdr(j)^2);
    x(j,:)=A1*sin(wn*t)+A2*cos(wn*t)+A3*cos(wdr(j)*t);
end
for k=1:3 % We could have used subplot this way all along.
    subplot(3,1,k)
    plot(t,x(k,:))
    title(['Response for wdr=',num2str(wdr(k)),',
    wn=',num2str(wn)])
    ylabel('Response x')
    grid
end
xlabel('Time, seconds')
```

Program 4-2: varywn.m

```
%This program is similar in form to varywdr.m, except that
%now the natural frequency is the variable.
%
%This file varies natural frequency of a single degree
%of freedom undamped oscillator at a set driving
%frequency. The initial displacement is 3, the velocity is
%1, the force magnitude per unit mass is 6, and the
%driving frequency is 7.
wdr=7;
x0=0;
v0=0;
f0=6;
tf=10;
wn=zeros(3,1);
x=zeros(3,1001);
for i=1:3;
    wn(i)=wdr; % Analagous to the initialization above.
    while wn(i)==wdr;
        wn(i)=input('Enter the natural frequency. ');
        if wn(i)==wdr;
            fprintf('This will produce resonance!!')
        end
    end
end
t=0:tf/1000:tf;
for j=1:3
    A1=v0/wn(j);
    A2=x0-(f0/(wn(j)^2-wdr^2));
    A3=f0/(wn(j)^2-wdr^2);
    x(j,:)=A1*sin(wn(j)*t)+A2*cos(wn(j)*t)+A3*cos(wdr*t);
end
for k=1:3
    subplot(3,1,k)
    plot(t,x(k,:))
    title(['Response for wdr=',num2str(wdr),',
    wn=',num2str(wn(k))])
    ylabel('Response x')
    grid
end
xlabel('Time, seconds')
```

Program 4-3: beatres.m

```
%The main feature of note for this code is the use of the
%if-then-else protocol in MATLAB to allow solutions other
%than “Inf” for the resonant case. By changing the values
%of the initial conditions, wn, and f0 to input
%statements, this program would become a general solver
%for the single-degree of freedom undamped oscillator,
%subject to harmonic forcing.
%
%This program shows beats and resonance in undamped
%systems. Again, in order to better see the effects, the
%initial velocity and displacement are zero.
x0=0;
v0=0;
wn=3;
wdr=input('Enter the driving frequency. ');
f0=6;
tf=120;
%This section chooses the proper response formula for
%the given situation.
t=0:tf/1000:tf;
if wdr==wn
    A1=v0/wn;
    A2=x0;
    A3=f0/2*wn;
    x=A1*sin(wn*t)+A2*cos(wn*t)+A3*t.*cos(wdr*t);
else
    A1=v0/wn;
    A2=x0-(f0/(wn^2-wdr^2));
    A3=f0/(wn^2-wdr^2);
    x=A1*sin(wn*t)+A2*cos(wn*t)+A3*cos(wdr*t);
end
plot(t,x);
if wdr==wn
    title('Example of Resonance Phenomenon');
else
    title('Example of Beat Phenomenon');
end
xlabel('Time, seconds')
ylabel('Response x')
grid
```

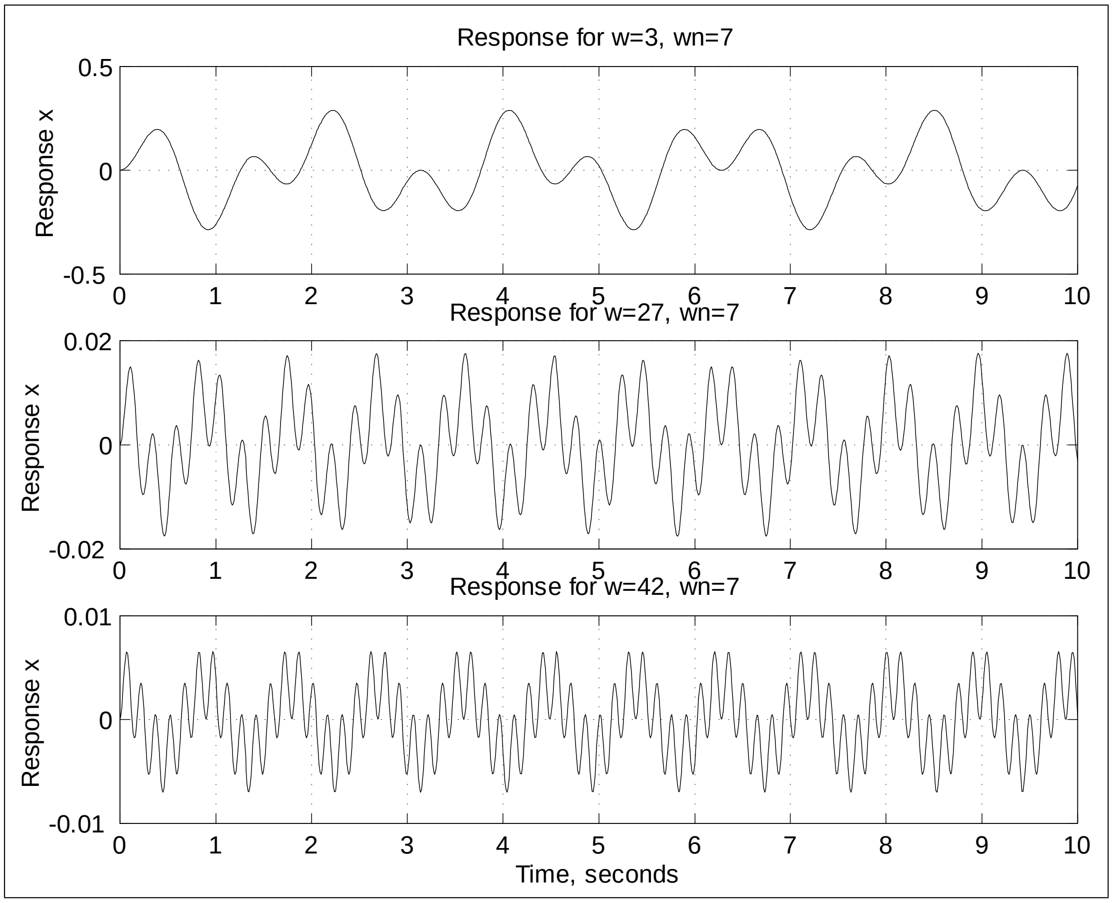

Figure 5.2: Vibration response for different driving frequencies.

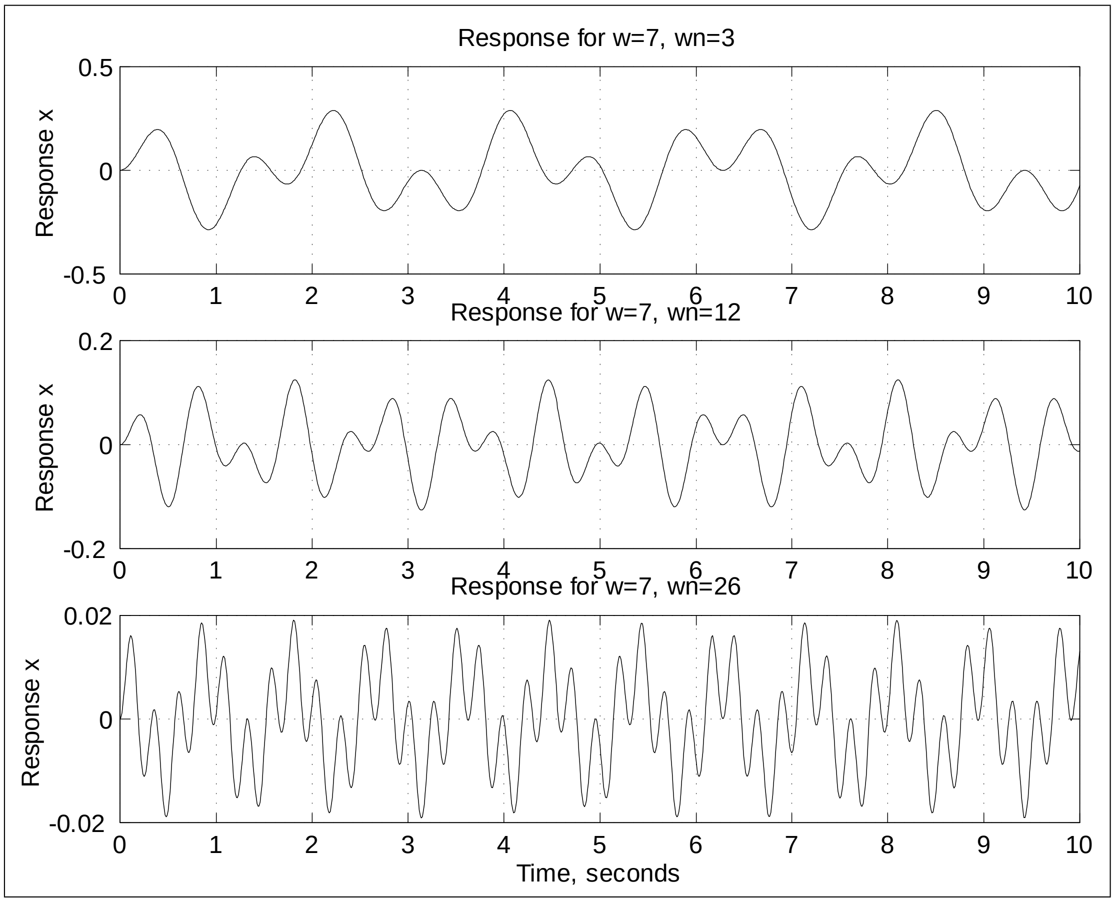

Figure 5.3: Vibration response for different natural frequencies.

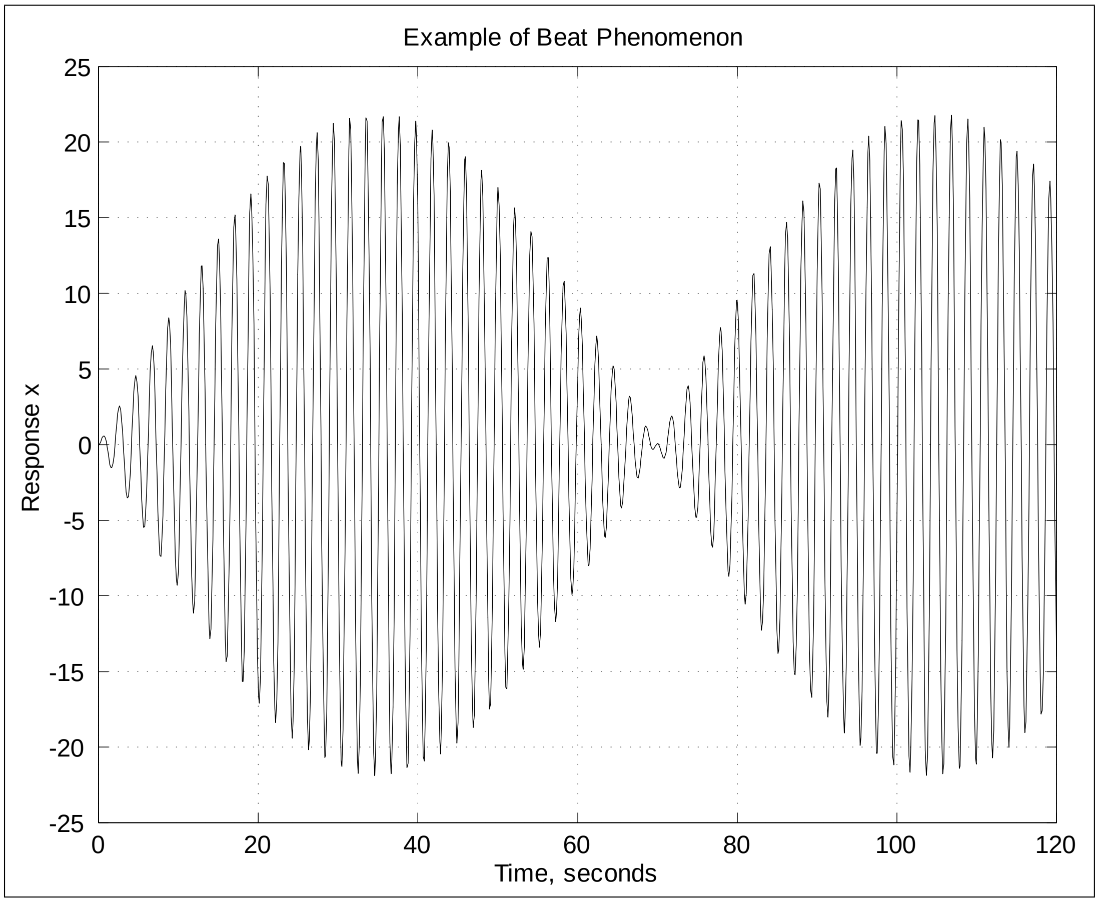

Figure 5.4: Example of the beating phenomenon.

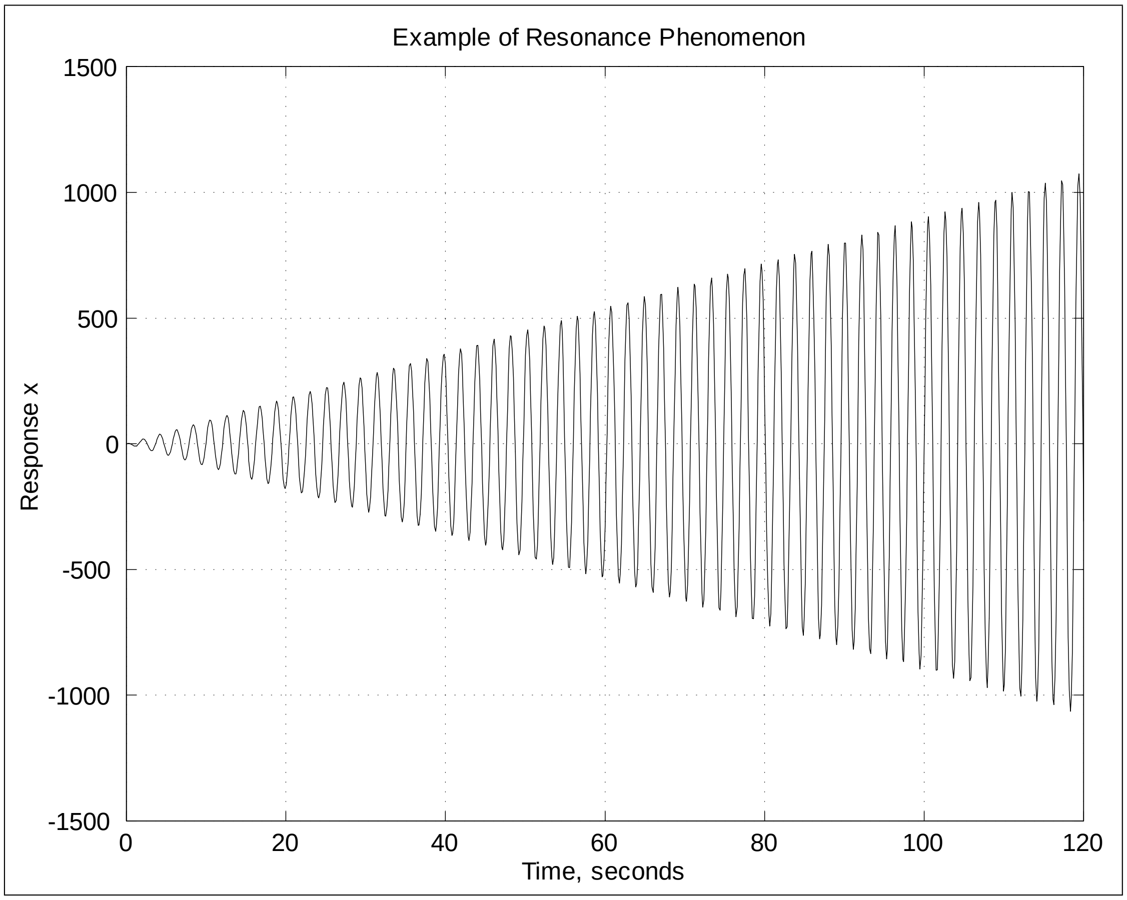

Figure 5.5: Example of resonant vibration.

Chapter 6

Harmonic Forcing of Damped SDOF Systems

Now, we will examine how the behavior of the system changes when we add damping. As with the unforced case, the equations of motion change when damping is added. They become:

$$m\ddot{x} + c\dot{x} + kx = F\cos(\omega t), \tag{6.1}$$

or with $\zeta = c/2m\omega_n$,

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = f\cos(\omega t), \tag{6.2}$$

where $f = F/m$. The homogeneous solution to this equation is of the form

$$x_h(t) = Ae^{-\zeta\omega t}\sin(\omega_d t + \theta), \tag{6.3}$$

where $\omega_d = \omega_n\sqrt{1-\zeta^2}$, and constants A, θ depend on initial conditions. The particular solution to the external force is

$$x_p(t) = A_0\cos(\omega t - \phi), \tag{6.4}$$

where

$$A_0 = \frac{f}{\sqrt{(\omega_n^2 - \omega^2) + (2\zeta\omega_n\omega)^2}},\ \phi = \tan^{-1}\frac{2\zeta\omega_n\omega}{\omega_n^2 - \omega^2}. \tag{6.5}$$

The complete solution, $x(t) = x_h(t)+x_p(t)$ is then used to evaluate constants A and θ. These constants were found for zero initial conditions using Maple; this solution is reflected in the code that follows.

The addition of damping causes the response of the system to differ slightly, as shown in Figures 6.1 and 6.2. In Figure 6.1, the damping ratio ζ was varied. This shows that the transient period of vibration varies inversely with damping ratio. The length of the transient period varies from about 4.5 seconds for $\zeta = 0.05$ to about 2 seconds for $\zeta = 0.3$, showing that, in many cases, the transient response can be ignored due to its short time period. However, for some cases, the transient period may be much longer or may have a very large amplitude, so it is always important to examine the transient effects of a system before neglecting them. Notice also that the damping ratio affects the amplitude of the steady-state vibration, also in an inverse relationship. That is, the amplitude of the response for $\zeta = 0.05$ is almost 2, while that for $\zeta = 0.3$ is less than 1.

Figure 6.2 shows the effects of changing the natural frequency. Notice how, for the two frequencies that are near the driving frequency, the transient period is quite long, almost 10 seconds. However, for the large natural frequency, the transient period is less than 4 seconds, which shows that the length of the transient period also depends on the natural frequency.

In the damped system, resonance also takes on a different meaning. Notice how, for $\omega = \omega_n$, the amplitude does not become infinite; the introduction of damping introduces a term that keeps the denominator of the steady-state amplitude from becoming zero. However, at this point, the phase angle becomes $90°$. For a damped system, this condition defines resonance, since it is also at this point that the denominator of the amplitude is a minimum. To prove this last assertion to yourself, look at the denominator of the amplitude constant, A_0. The amplitude will be maximized when the denominator is minimized. Both terms are never negative, so the minimum will occur when the two frequencies are equal (making the first term of the denominator zero). Also, as the driving frequency increases greatly, the amplitude nears zero.

The MATLAB code below was used to produce the figures. Instead of inserting a `while` statement to control the range of damping ratios used, we use an `if` statement, in tandem with an `error` statement. If the damping ratio is outside the allowed range, MATLAB will halt execution of the program, printing the statement found inside the quotation marks to the screen. Controlling input in this manner is more drastic than the method used in the previous programs. However, those programs had many more values to input. Forcing the user of a program to reenter several values because of a typo seems a harsh penalty to this programmer. Also, this program introduces MATLAB's method for continuing lines. To continue an expression onto the next line, simply type three periods in succession (ellipsis) and move to the next line. This is helpful for long mathematical expressions, like the constants of integration below. This ellipsis can be placed anywhere a space would be allowed, so continuing in the middle of a variable name, for example, is not recommended.

Program 5-1: harmzeta.m

```
%This program solves for the response of
%a damped single degree of freedom system subject to
%a harmonic external force. The expressions used for the
%constants were found by using MAPLE.
%
wdr=3;
wn=3.5;
fo=4;
tf=10;
t=0:tf/1000:tf;
for k=1:3
    zeta(k)=input('Enter a damping ratio (zeta). ');
    if (zeta(k)<0 | zeta(k)>=1)
        error('Zeta out of range for this program!')
    end
end
for k=1:3
    wd=wn*sqrt(1-zeta(k)^2);
    Ao=fo/sqrt((wn^2-wdr^2)^2+(2*zeta(k)*wn*wdr)^2);
    phi=atan2(2*zeta(k)*wn*wdr,(wn^2-wdr^2));
    Z1=-zeta(k)*wn-wdr*tan(phi);
    Z2=sqrt((zeta(k)*wn)^2+2*zeta(k)*wn*wdr*tan(phi)+ ... % Continu-
ation.
        (wdr*tan(phi))^2+wd^2); % The extra tab is not necessary, but
helpful.
    Z=(Z1+Z2)/wd;
    Anum=Ao*((zeta(k)*wn*Z-wd)*cos(phi)+wdr*Z*sin(phi));
    Aden=Z*wd;
    A=Anum/Aden;
    theta=2*atan(Z);
    x(k,:)=A*exp(-zeta(k)*wn*t).*sin(wd*t+theta)+Ao*cos(wdr*t-phi);
end
for k=1:3
    subplot(3,1,k)
    plot(t,x(k,:))
    title(['Response for zeta=',num2str(zeta(k)),', wn=', ...
        num2str(wn),', and wdr=', num2str(wdr)])
    ylabel('Response x')
    grid
end
xlabel('Time, seconds')
```

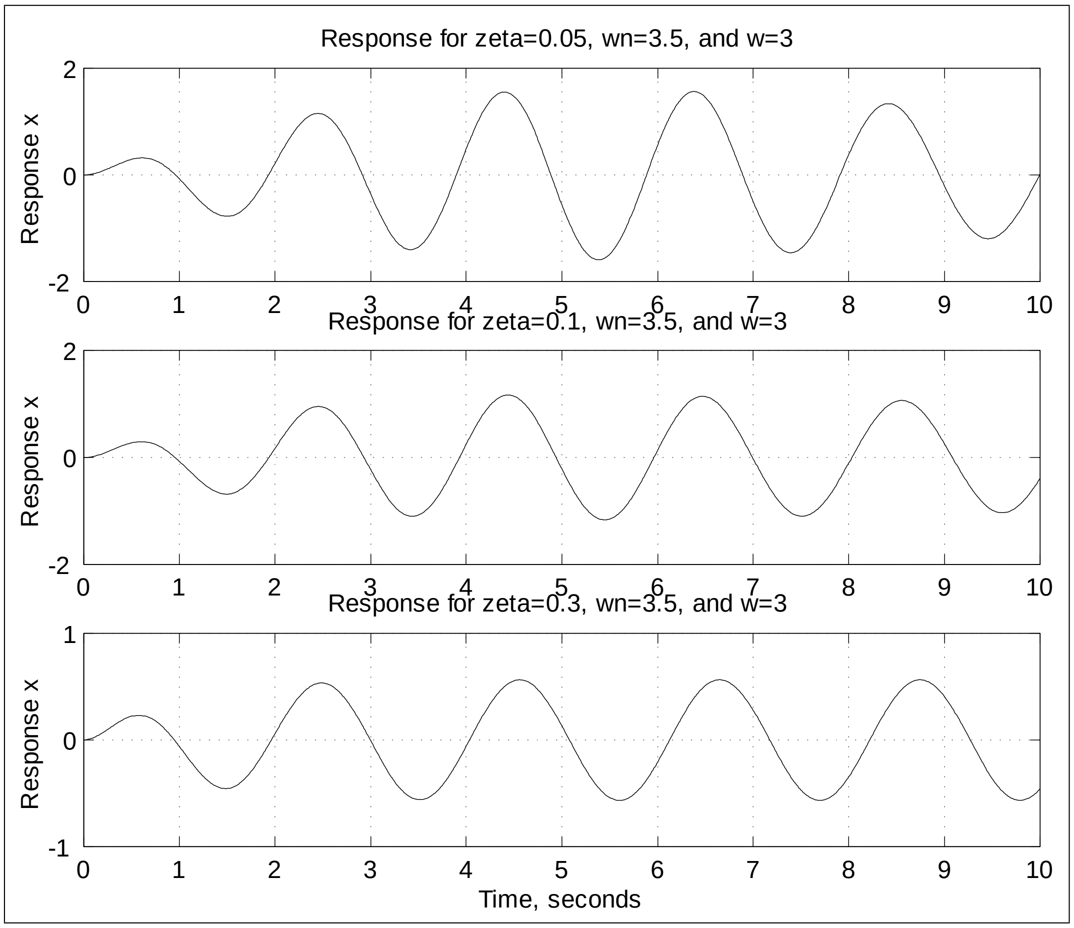

Figure 6.1: Damped response for three different damping ratios.

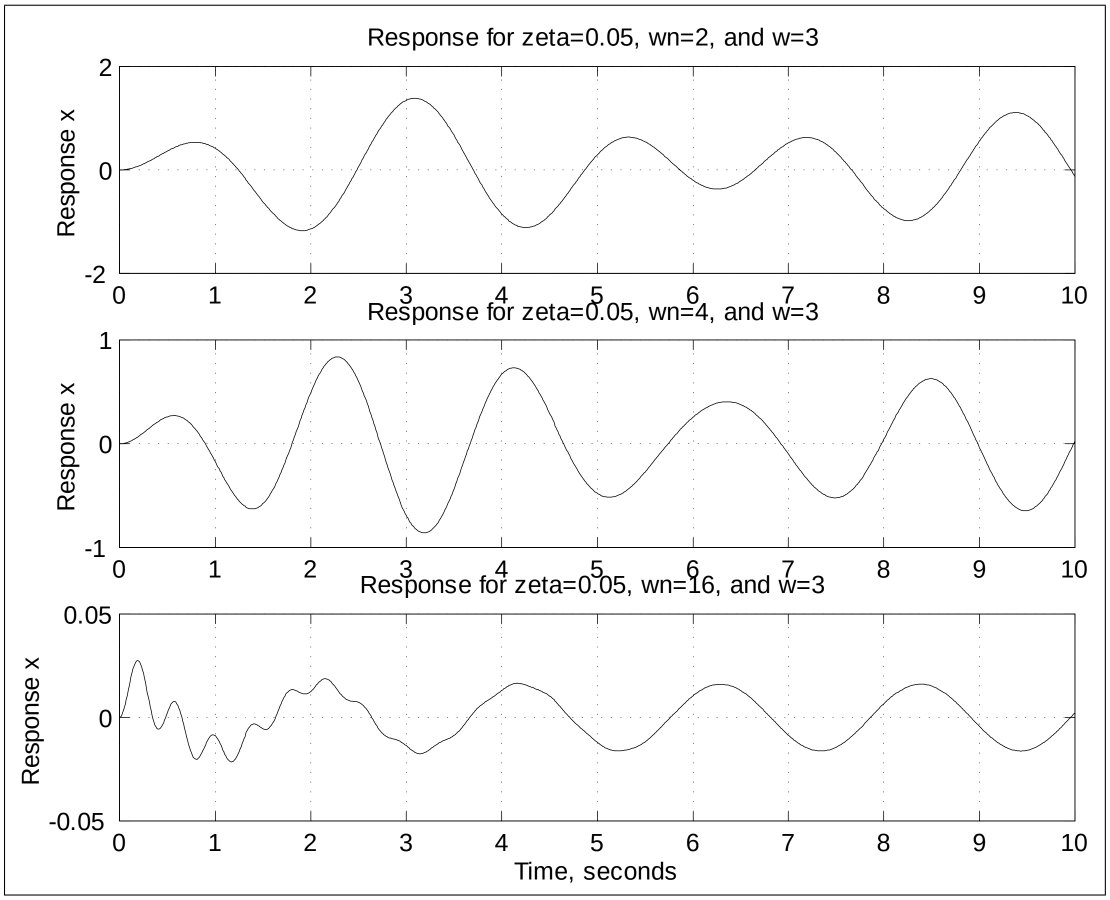

Figure 6.2: Response variation with different natural frequencies.

Chapter 7

Base Excitation of SDOF Systems

The base excitation problem is illustrated in Figure 7.1. Let the motion of the base be denoted by $y(t)$ and the response of the mass by $x(t)$, and assuming that the base has harmonic motion of the form $y(t) = Y\sin(\omega_b t)$. The equation of motion for this system is:

$$m\ddot{x} + c(\dot{x} - \dot{y}) + k(x - y) = 0. \tag{7.1}$$

Using the assumed form for the motion, we can substitute for y and its derivative, resulting in:

$$m\ddot{x} + c\dot{x} + kx = cY\omega_b \cos\omega_b t + kY\sin\omega_b t, \tag{7.2}$$

which when divided through by the mass, yields

$$\ddot{x} + 2\zeta\omega\dot{x} + \omega^2 x = 2\zeta\omega\omega_b \cos\omega_b t + \omega^2 Y\sin\omega_b t. \tag{7.3}$$

The homogeneous solution is of the form:

$$x_h = Ae^{-\zeta\omega t}\sin(\omega_d t + \theta). \tag{7.4}$$

The expression for each part of the particular solution is similar to that for the general sinusoidal forcing function; the sine term produces a sine solution, and the cosine term produces a cosine solution. If we find these solutions and combine their sum into a single sinusoid, we obtain:

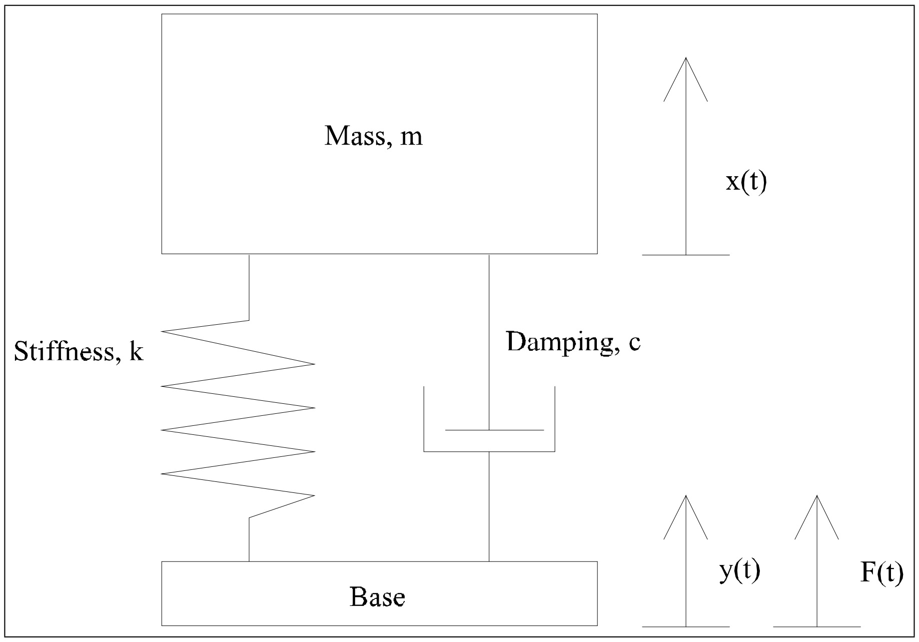

Figure 7.1: Typical single degree of freedom system subject to base excitation.

$$x_p = A_o \cos(\omega_b t - \phi_1 - \phi_2), \tag{7.5}$$

where

$$A_o = \omega Y \sqrt{\frac{\omega^2 + (2\zeta\omega_b)^2}{(\omega^2 - \omega_b^2)^2 + (2\zeta\omega\omega_b)^2}},\ \phi_1 = \tan^{-1}\frac{2\zeta\omega\omega_b}{\omega^2 - \omega_b^2},\ \phi_2 = \tan^{-1}\frac{\omega}{2\zeta\omega_b}.$$

Thus, the complete solution is the sum of the homogeneous and particular solutions, or:

$$x(t) = Ae^{-\zeta\omega t}\sin(\omega_d t + \theta) + A_o\cos(\omega_b t - \phi_1 - \phi_2). \tag{7.6}$$

This equation tells us a great deal about the motion of the mass. First, we can see that the particular solution represents the steady-state response, while the homogeneous solution is the transient response, since the particular solution is independent of the initial displacement and velocity. Using MAPLE to solve the initial value problem (with given initial velocity and displacement not necessarily equal to zero), we find that both are dependent upon the initial velocity and displacement. However, the expression for the constants A and θ is, in general, very difficult to solve, even for MAPLE. So, for the sake of programming in MATLAB, the initial velocity and displacement were both assumed to be zero. This assumption yielded simpler equations, which were used in the base excitation programs which follow.

Figure 7.2 shows the effects of changing the excitation frequency while holding all other parameters constant. In the steady state, from about three seconds forward, note that the frequency of vibration increases with the base frequency. This is expected, since the base excitation portion dominates the steady state. Of particular note is the bottom plot, with $\omega_b = 20$. In the transient portion, the response has the shape of a sum of two sinusoids; these are, of course, the transient and steady-state functions. Since the base excitation is of such high frequency, this graph shows best what is happening between the transient and steady responses. Note that, if a line was drawn through the upper or lower peaks of the motion, the result would be a curve similar to that exhibited by a damped free response. The midpoint of the oscillation caused by the steady response becomes exponentially closer to zero with increasing time, as the transient response diminishes.

Figure 7.3 gives plots for three different vibration amplitudes. The differences caused by changing the amplitude is what would be expected; the maximum amplitude of the overall vibration and of the steady-state response both increase with increasing input amplitude.

The plots in Figure 7.4 for various damping ratios show two effects of changing the damping ratio. First, the change in damping ratio causes the length of the transient period to vary; an increase in ζ causes the transient period to decrease, as the plots show. Also, the change in damping ratio causes a change in the frequency of the transient vibration. Again, an increase in ζ causes a decrease in the damped natural frequency, although the decrease is not entirely evident from just looking at the plots. This is because the plots also include the base excitation (steady-state) terms, whose frequency has not changed.

The MATLAB code for this situation is much more complex than the code used in previous examples. This is mainly due to the increased difficulty encountered when the external load is applied to a part of the structure. When this approach is used, the relative displacement of the two parts of the structure becomes the important factor, instead of the displacement of the structure relative to some ground. The code for this example uses a solution obtained from the MAPLE solve routine to produce a plot of the response. An important consideration to note is that the answer given by MAPLE results in two possible solutions; the MATLAB code attempts to locate the correct one. The way in which MATLAB chooses the correct solution is to check which of them matches the initial displacement condition. Note from the plots that the initial displacements are zero for all plots, as are the initial velocities, so we can be confident in the plotted solutions. The three programs given below show the changes that need to be made in a program in order to test different parameters.

Program 6-1: varywb.m

```
%This program solves the base excitation problem. The
%code assumes a sinusoidal base function. Also, the
%program tests three different natural frequencies.
%
y0=input('Enter the base excitation magnitude. ');
zeta=input('Enter the damping ratio (zeta). ');
if (zeta<0 | zeta>=1) % The usual test on damping ratio.
    error('Damping ratio not in acceptable range!')
end
wn=4;
tf=10;
t=0:tf/1000:tf;
for k=1:3
    wb(k)=input('Enter a base excitation frequency. ');
end
for m=1:3
%
%This section solves the transient response, using the
%equations obtained from MAPLE.
%
    wd=wn*sqrt(1-zeta^2);
    phi1=atan2(2*zeta*wn*wb(m),(wn^2-wb(m)^2));
    phi2=atan2(wn,2*zeta*wb(m));
    xi=phi1+phi2;
%These constants are what produces the two possible
%solutions discussed above. Notice the way by which
%the extraordinarily long expressions for the constants
%are broken into parts, to keep the expressions from
%spreading over several lines.
    Z1=(-zeta*wn-wb(m)*tan(xi)+sqrt((zeta*wn)^2+2*zeta* ...
        wn*wb(m)*tan(xi)+(wb(m)*tan(xi))^2+wd^2))/wd;
    Z2=(-zeta*wn-wb(m)*tan(xi)-sqrt((zeta*wn)^2+2*zeta* ...
        wn*wb(m)*tan(xi)+(wb(m)*tan(xi))^2+wd^2))/wd;
    Anum=sqrt((wn^2+(2*zeta*wb(m))^2)/((wn^2-wb(m)^2)^2+(2* ...
        zeta*wb(m)*wn)^2))*wn*y0;
    Bnum1=(-wd*cos(xi)+Z1*zeta*wn*cos(xi)+Z1*wb(m)*sin(xi));
    Bnum2=(-wd*cos(xi)+Z2*zeta*wn*cos(xi)+Z2*wb(m)*sin(xi));
    Aden1=wd*Z1;
    Aden2=wd*Z2;
    A1=Anum*Bnum1/Aden1;
    A2=Anum*Bnum2/Aden2;
    th1=2*atan(Z1);
    th2=2*atan(Z2);
    y1(m,:)=A1*exp(-zeta*wn*t).*sin(wd*t+th1);
    y2(m,:)=A2*exp(-zeta*wn*t).*sin(wd*t+th2);
```

```
end
%This portion solves the steady-state response.
    for j=1:3
    A=sqrt((wn^2+(2*zeta*wb(j))^2)/((wn^2-wb(j)^2)^2+(2*zeta* ...
    wn*wb(j))^2));
    phi1=atan2(2*zeta*wn*wb(j),(wn^2-wb(j)^2));
    phi2=atan2(wn,(2*zeta*wb(j)));
    xp(j,:)=wn*y0*A*cos(wb(j)*t-phi1-phi2);
end
if (xp(1,1)+y1(1,1)==xp(2,1)+y1(2,1)==xp(3,1)+y1(3,1)==0)
    x=xp+y1;
else
    x=xp+y2;
end
for i=1:3
    subplot(3,1,i)
    plot(t,x(i,:))
    ylabel('Response x');
    title(['Base Excitation with wb=',num2str(wb(i)), ...
    ' and wn=',num2str(wn)]);
    grid
end
xlabel('Time, seconds')
```

Program 6-2: varyyobe.m

```
%This program solves the base excitation problem. The
%code assumes a sinusoidal base function. Also, the
%program tests three different base amplitudes.
%Notice that the natural frequency is now given as
%a set variable, and the variable 'y0' is a user-input matrix.
%
wb=input('Enter the base excitation frequency. ');
zeta=input('Enter the damping ratio (zeta). ');
if (zeta<0 | zeta>=1) % The usual test on damping ratio.
        error('Damping ratio not in acceptable range!')
end
wn=4;
tf=10;
t=0:tf/1000:tf;
for k=1:3
    y0(k)=input('Enter a base excitation magnitude. ');
end
for m=1:3
    wd=wn*sqrt(1-zeta^2);
    phi1=atan2(2*zeta*wn*wb,(wn^2-wb^2));
    phi2=atan2(wn,2*zeta*wb);
    xi=phi1+phi2;
    Z1=(-zeta*wn-wb*tan(xi)+sqrt((zeta*wn)^2+2*zeta* ...
        wn*wb*tan(xi)+(wb*tan(xi))^2+wd^2))/wd;
    Z2=(-zeta*wn-wb*tan(xi)-sqrt((zeta*wn)^2+2*zeta* ...
        wn*wb*tan(xi)+(wb*tan(xi))^2+wd^2))/wd;
    Anum=sqrt((wn^2+(2*zeta*wb)^2)/((wn^2-wb^2)^2+(2* ...
        zeta*wb*wn)^2))*wn*y0(m);
    Bnum1=(-wd*cos(xi)+Z1*zeta*wn*cos(xi)+Z1*wb*sin(xi));
    Bnum2=(-wd*cos(xi)+Z2*zeta*wn*cos(xi)+Z2*wb*sin(xi));
    Aden1=wd*Z1; Aden2=wd*Z2;
    A1=Anum*Bnum1/Aden1;
    A2=Anum*Bnum2/Aden2;
    th1=2*atan(Z1);
    th2=2*atan(Z2);
    y1(m,:)=A1*exp(-zeta*wn*t).*sin(wd*t+th1);
    y2(m,:)=A2*exp(-zeta*wn*t).*sin(wd*t+th2);
end
for j=1:3
    A=sqrt((wn^2+(2*zeta*wb)^2)/((wn^2-wb^2)^2+(2*zeta* ...
        wn*wb)^2));
    phi1=atan2(2*zeta*wn*wb,(wn^2-wb^2));
    phi2=atan2(wn,(2*zeta*wb));
    xp(j,:)=wn*y0(j)*A*cos(wb*t-phi1-phi2);
end
```

```
if (xp(1,1)+y1(1,1)==xp(2,1)+y1(2,1)==xp(3,1)+y1(3,1)==0)
      x=xp+y1;
else
      x=xp+y2;
end
for i=1:3
      subplot(3,1,i)
      plot(t,x(i,:))
      ylabel('Response x');
      title(['Base Excitation with wb=',num2str(wb), ...
      ', wn=',num2str(wn),', and y0=',num2str(y0(i))]);
      grid
end
xlabel('Time, seconds')
```

Program 6-3: varyzbe.m

```
%This program solves the base excitation problem. The
%code assumes a sinusoidal base function. Also, the
%program tests three different damping ratios. Again, note
%the changes between this program and the previous one.
%
y0=input('Enter the base excitation magnitude. ');
wb=input('Enter the base excitation frequency. ');
wn=4;
tf=10;
t=0:tf/1000:tf;
for k=1:3
    zeta(k)=input('Enter a damping ratio (zeta). ');
    if (zeta(k)<0 | zeta(k)>=1) % The usual test on damping ratio.
        error('Damping ratio not in acceptable range!')
    end
end
for m=1:3
    wd=wn*sqrt(1-zeta(m)^2);
    phi1=atan2(2*zeta(m)*wn*wb,(wn^2-wb^2));
    phi2=atan2(wn,2*zeta(m)*wb);
    xi=phi1+phi2;
    Z1=(-zeta(m)*wn-wb*tan(xi)+sqrt((zeta(m)*wn)^2+2*zeta(m)* ...
        wn*wb*tan(xi)+(wb*tan(xi))^2+wd^2))/wd;
    Z2=(-zeta(m)*wn-wb*tan(xi)-sqrt((zeta(m)*wn)^2+2*zeta(m)* ...
        wn*wb*tan(xi)+(wb*tan(xi))^2+wd^2))/wd;
    Anum=sqrt((wn^2+(2*zeta(m)*wb)^2)/((wn^2-wb^2)^2+(2* ...
        zeta(m)*wb*wn)^2))*wn*y0;
    Bnum1=(-wd*cos(xi)+Z1*zeta(m)*wn*cos(xi)+Z1*wb*sin(xi));
    Bnum2=(-wd*cos(xi)+Z2*zeta(m)*wn*cos(xi)+Z2*wb*sin(xi));
    Aden1=wd*Z1;
    Aden2=wd*Z2;
    A1=Anum*Bnum1/Aden1;
    A2=Anum*Bnum2/Aden2;
    th1=2*atan(Z1);
    th2=2*atan(Z2);
    y1(m,:)=A1*exp(-zeta(m)*wn*t).*sin(wd*t+th1);
    y2(m,:)=A2*exp(-zeta(m)*wn*t).*sin(wd*t+th2);
end
for j=1:3
    A=sqrt((wn^2+(2*zeta(j)*wb)^2)/((wn^2-wb^2)^2+(2*zeta(j)* ...
        wn*wb)^2));
    phi1=atan2(2*zeta(j)*wn*wb,(wn^2-wb^2));
    phi2=atan2(wn,(2*zeta(j)*wb));
    xp(j,:)=wn*y0*A*cos(wb*t-phi1-phi2);
end
```

```
if (xp(1,1)+y1(1,1)==xp(2,1)+y1(2,1)==xp(3,1)+y1(3,1)==0)
    x=xp+y1;
else
    x=xp+y2;
end
for i=1:3
    subplot(3,1,i)
    plot(t,x(i,:))
    ylabel('Response x');
    title(['Base Excitation with wb=',num2str(wb), ...
        ' and zeta=',num2str(zeta(i))]);
grid
end
xlabel('Time, seconds')
```

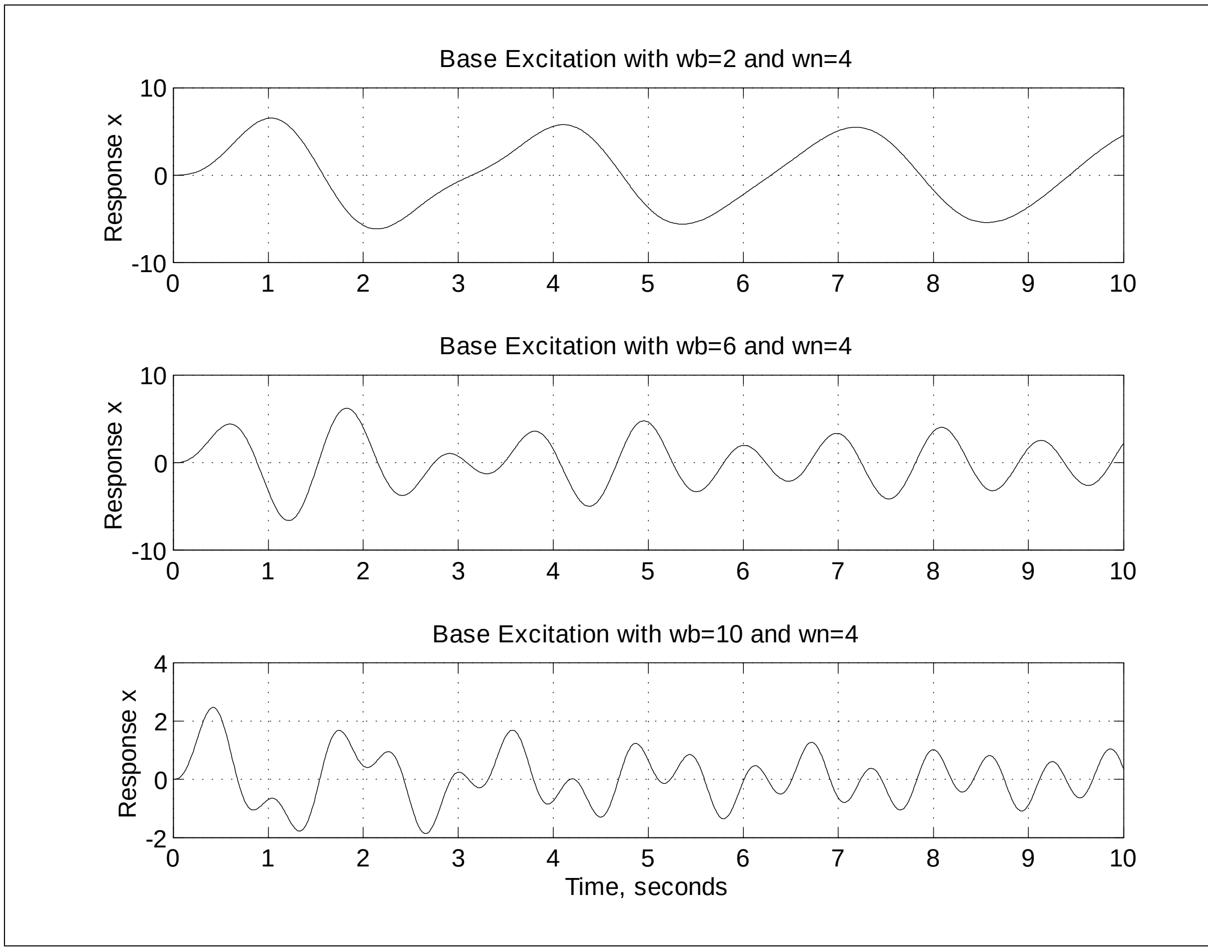

Figure 7.2: Response of a base-excited system subject to three different excitation frequencies.

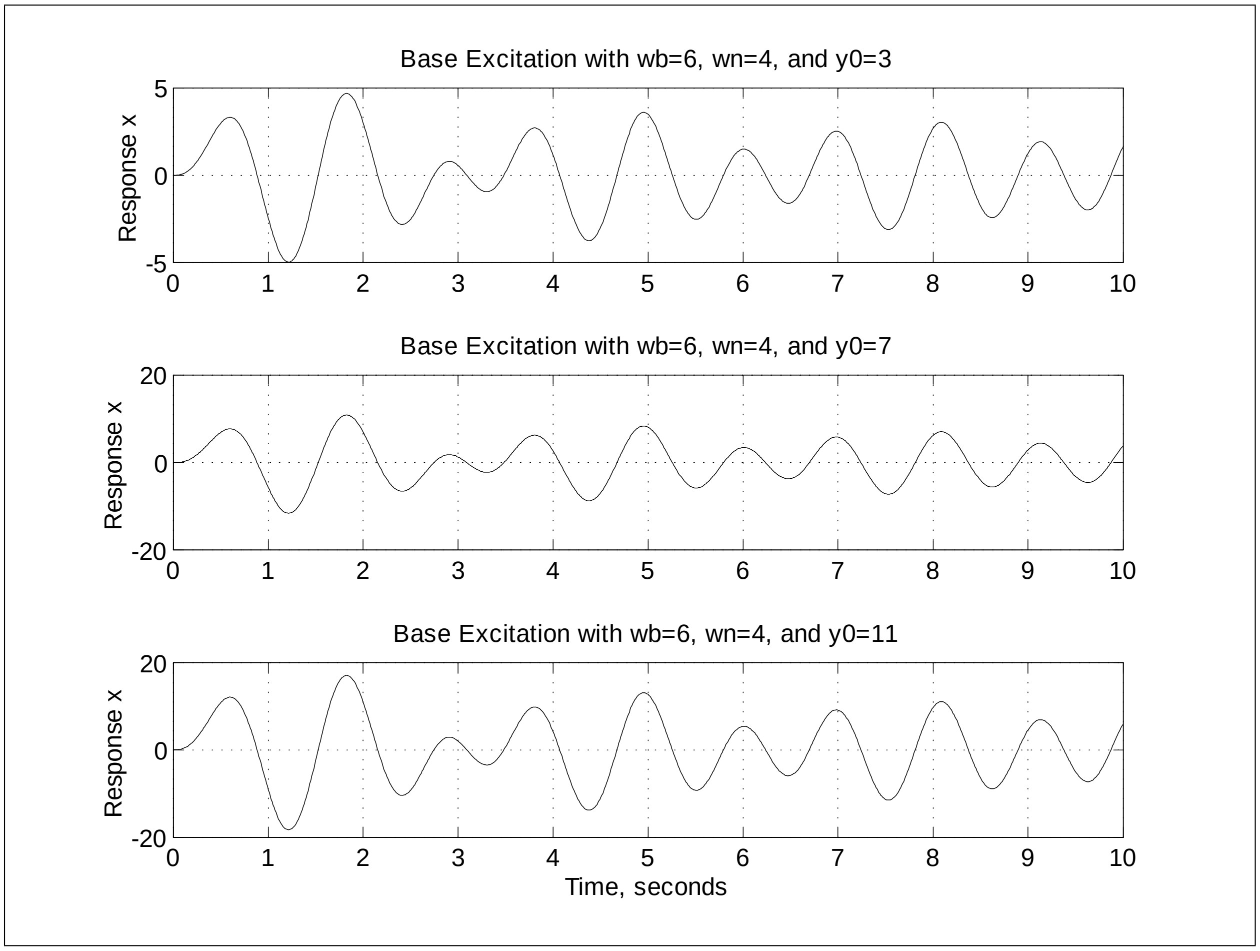

Figure 7.3: Results obtained for three different base excitation magnitudes.

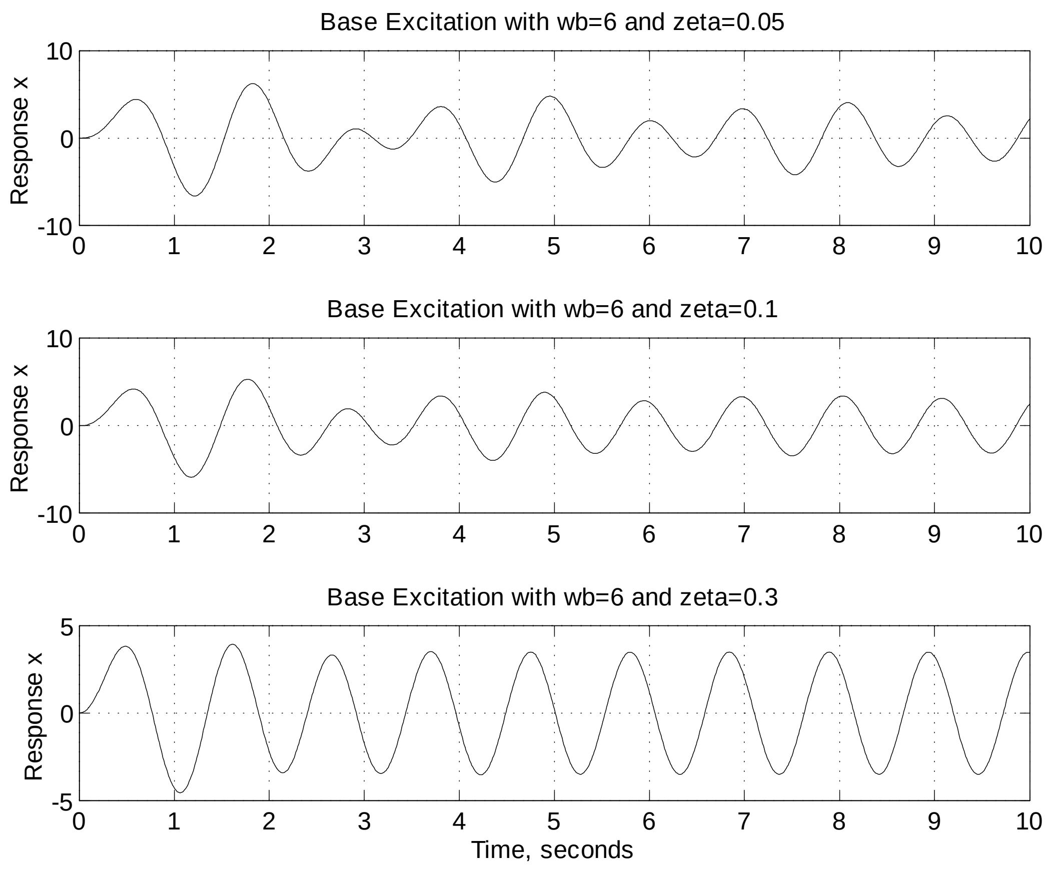

Figure 7.4: Response of a base-excited system for different damping ratios.

Chapter 8

SDOF Systems with a Rotating Unbalance

A rotating unbalance is depicted in Figure 8.1. Assume that the guides are frictionless. The radius e is measured from the center of mass of the mass m. To construct the equation of motion, we need an expression for the motion of the rotating unbalance in terms of x. If the mass rotates with a constant angular velocity ω_r, then the circle it defines can be described parametrically as:

$$x(t) = e \sin \omega_r t, \; y(t) = e \cos \omega_r t. \tag{8.1}$$

Note that the sine defines the x coordinate because in our chosen coordinates, x is vertical. Having this expression for x, we can then construct the equations of motion. The position coordinate of the rotating unbalance is $x + \sin \omega_r t$, and the acceleration is the second derivative of this expression with respect to time. The acceleration of the mass without the unbalance is $\ddot{x}$. Adding in the effects of the stiffness and damper, we get:

$$(m - m_o)\,\ddot{x} + m_o \frac{d^2}{dt^2}\left(x + e \sin \omega_r t\right) = -kx - c\dot{x}, \tag{8.2}$$

or

$$(m - m_o)\,\ddot{x} + m_o \left(\ddot{x} - e\omega_r^2 \sin \omega_r t\right) = -kx - c\dot{x}. \tag{8.3}$$

Finally, collecting x and its derivatives, moving the sine term to the other side of the expression, and dividing by the system mass gives the final equation of motion:

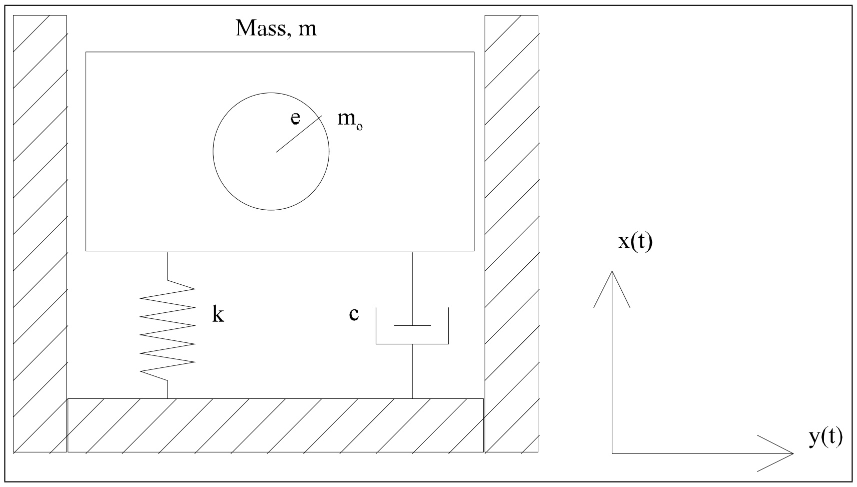

Figure 8.1: Schematic of the system in question (note the coordinate axes chosen).

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = m_o e\omega_r^2 \sin\omega_r t. \tag{8.4}$$

Note that this is identical to the harmonic forcing function case we encountered earlier, except that now our force is in the form of a sine rather than a cosine. For that reason, the particular solution is of the form:

$$x_p(t) = X\sin(\omega_r t - \phi), \tag{8.5}$$

where, with $r = \omega_r/\omega_n$,

$$X = \frac{m_o e}{m}\frac{r^2}{\sqrt{(1-r^2)^2 + (2\zeta r)^2}}, \ \phi = \tan^{-1}\frac{2\zeta r}{1-r^2}. \tag{8.6}$$

As before, the homogenous solution for this expression is:

$$x_h(t) = Ae^{-\zeta\omega_n t}\sin(\omega_d t + \theta), \tag{8.7}$$

where A and θ are determined from the initial conditions. The final solution is then $x(t) = x_p(t) + x_h(t)$.

For the purpose of creating a program in MATLAB, the initial conditions were assumed to be zero. Then, MAPLE was used to solve the resulting initial value problem. The solution to this is not reproduced here, due to the complexity of the expression; the solution for A and θ depend on the solution to a quadratic equation. The MATLAB code that follows contains the expression (in a few parts) for the constants in question.

Figures 8.2 through 8.4 show different varying parameter sets for the system. Unless otherwise specified, $m = 7$, $m_o = 3$, and $e = 0.1$. For Figure 8.2, the natural frequency was varied while holding all other parameters constant. Notice how, when ω_n is not a multiple of ω_r, the motion is the sum of two sinusoids; this is shown best by the top plot, where $\omega_n = 2$. For the highest natural frequency tested, the oscillation occurs along a single sinusoid. This is because the natural frequency of the system is too high to be excited by the relatively slow rotation frequencies. The first two plots have natural frequencies small enough to be excited by the slow rotation of the eccentric mass.

In Figure 8.3, the system damping is varied. The result is that the transient portion (the portion with the curve that looks like the sum of sinusoids) becomes smaller, to the point where it disappears at $\zeta = 0.3$. A difference in the magnitude of oscillation, as would be predicted from the expression we have derived for the parameter X, is not present because the frequency ratio we are testing is in the range where oscillation magnitude shows little variation with damping ratio. This consideration is important in the design of machinery; if the machine can be designed to have a much higher natural frequency than the oscillating mass, then the level of damping can be made low without increasing the amplitude past acceptable levels.

Finally, Figure 8.4 shows the variation of vibration with increasing system mass. Notice how the amplitude of the vibration decreases with increasing mass; this is due to the dependence of X on m_o/m. As the mass ratio decreases, so does the amplitude of vibration.

Program 7-1: vrywnrot.m

```
%This program solves for the response of a single
%degree of freedom system having a rotating unbalance.
%The equations of motion were derived using MAPLE and
%are valid only for zero initial conditions.
%
mo=3;
m=7;
e=0.1;
wr=4;
zeta=0.05;
tf=10;
t=0:tf/1000:tf;
for i=1:3
wn(i)=input('Enter a natural frequency. ');
wd(i)=wn(i)*sqrt(1-zeta^2);
end
for j=1:3
r=wr/wn(j);
X=mo*e/m*(r^2/sqrt((1-r^2)^2+(2*zeta*r)^2));
phi=atan2(2*zeta*r,(1-r^2));
Z1=(-zeta*wn(j)+wr*cot(phi))/wd(j);
Z2=sqrt((zeta*wn(j))^2-2*zeta*wn(j)*wr*cot(phi)+ ...
(wr*cot(phi))^2+wd(j)^2)/wd(j);
Z=Z1+Z2;
theta=2*atan(Z);
Anum=X*(wd(j)*sin(phi)-Z*zeta*wn(j)*sin(phi)+Z*wr*cos(phi));
Aden=Z*wd(j);
A=Anum/Aden;
xh(j,:)=A*exp(-zeta*wn(j)*t).*sin(wd(j)*t+theta);
xp(j,:)=X*sin(wr*t-phi);
end
x=xp+xh;
for k=1:3
subplot(3,1,k)
plot(t,x(k,:))
title(['Rotating Unbalance with wr=',num2str(wr),' wn=', ...
num2str(wn(k)),' and zeta=',num2str(zeta)]);
grid
end
xlabel('Time, seconds')
```

Program 7-2: vryzrot.m

```
%This program solves for the response of a single
%degree of freedom system having a rotating unbalance.
%The equations of motion were derived using MAPLE and
%are valid only for zero initial conditions.
%
mo=3;
m=7;
e=0.1;
wr=4;
wn=12;
tf=10;
t=0:tf/1000:tf;
for i=1:3
zeta(i)=input('Enter a damping ratio (zeta). ');
end
for j=1:3
wd=wn*sqrt(1-zeta(j)^2);
r=wr/wn;
X=mo*e/m*(r^2/sqrt((1-r^2)^2+(2*zeta(j)*r)^2));
phi=atan2(2*zeta(j)*r,(1-r^2));
Z1=(-zeta(j)*wn+wr*cot(phi))/wd;
Z2=sqrt((zeta(j)*wn)^2-2*zeta(j)*wn*wr*cot(phi)+ ...
(wr*cot(phi))^2+wd^2)/wd;
Z=Z1+Z2;
theta=2*atan(Z);
Anum=X*(wd*sin(phi)-Z*zeta(j)*wn*sin(phi)+Z*wr*cos(phi));
Aden=Z*wd;
A=Anum/Aden;
xh(j,:)=A*exp(-zeta(j)*wn*t).*sin(wd*t+theta);
xp(j,:)=X*sin(wr*t-phi);
end
x=xp+xh;
for k=1:3
subplot(3,1,k)
plot(t,x(k,:))
title(['Rotating Unbalance with wr=',num2str(wr),' wn=', ...
num2str(wn),' and zeta=',num2str(zeta(k))]);
ylabel('Response x')
grid
end
xlabel('Time, seconds')
```

Program 7-3: vrymrot.m

```
%This program solves for the response of a single
%degree of freedom system having a rotating unbalance.
%The equations of motion were derived using MAPLE and
%are valid only for zero initial conditions.
%
mo=3;
zeta=0.05;
e=0.1;
wr=4;
wn=12;
tf=10;
t=0:tf/1000:tf;
for i=1:3
m(i)=input('Enter a system mass. ');
end
for j=1:3
wd=wn*sqrt(1-zeta^2);
r=wr/wn;
X=mo*e/m(j)*(r^2/sqrt((1-r^2)^2+(2*zeta*r)^2));
phi=atan2(2*zeta*r,(1-r^2));
Z1=(-zeta*wn+wr*cot(phi))/wd;
Z2=sqrt((zeta*wn)^2-2*zeta*wn*wr*cot(phi)+ ...
(wr*cot(phi))^2+wd^2)/wd;
Z=Z1+Z2;
theta=2*atan(Z);
Anum=X*(wd*sin(phi)-Z*zeta*wn*sin(phi)+Z*wr*cos(phi));
Aden=Z*wd;
A=Anum/Aden;
xh(j,:)=A*exp(-zeta*wn*t).*sin(wd*t+theta);
xp(j,:)=X*sin(wr*t-phi);
end
x=xp+xh;
for k=1:3
subplot(3,1,k)
plot(t,x(k,:))
title(['Rotating Unbalance with wr=',num2str(wr),', wn=', ...
num2str(wn),', mass=',num2str(m(k)),', and rotating mass=', ...
num2str(mo)]);
grid
end
xlabel('Time, seconds')
```

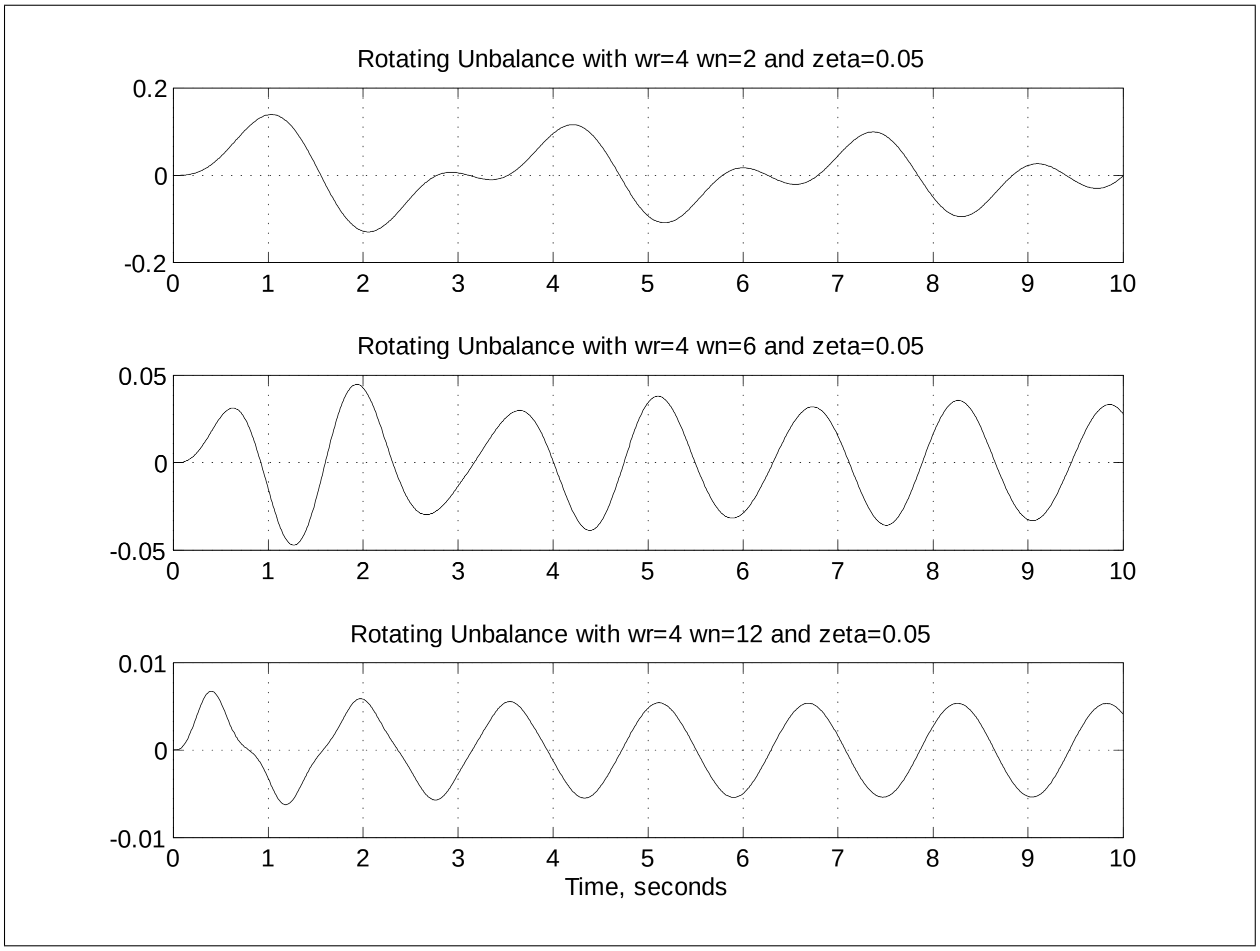

Figure 8.2: Response of systems with different natural frequencies to a rotating unbalance.

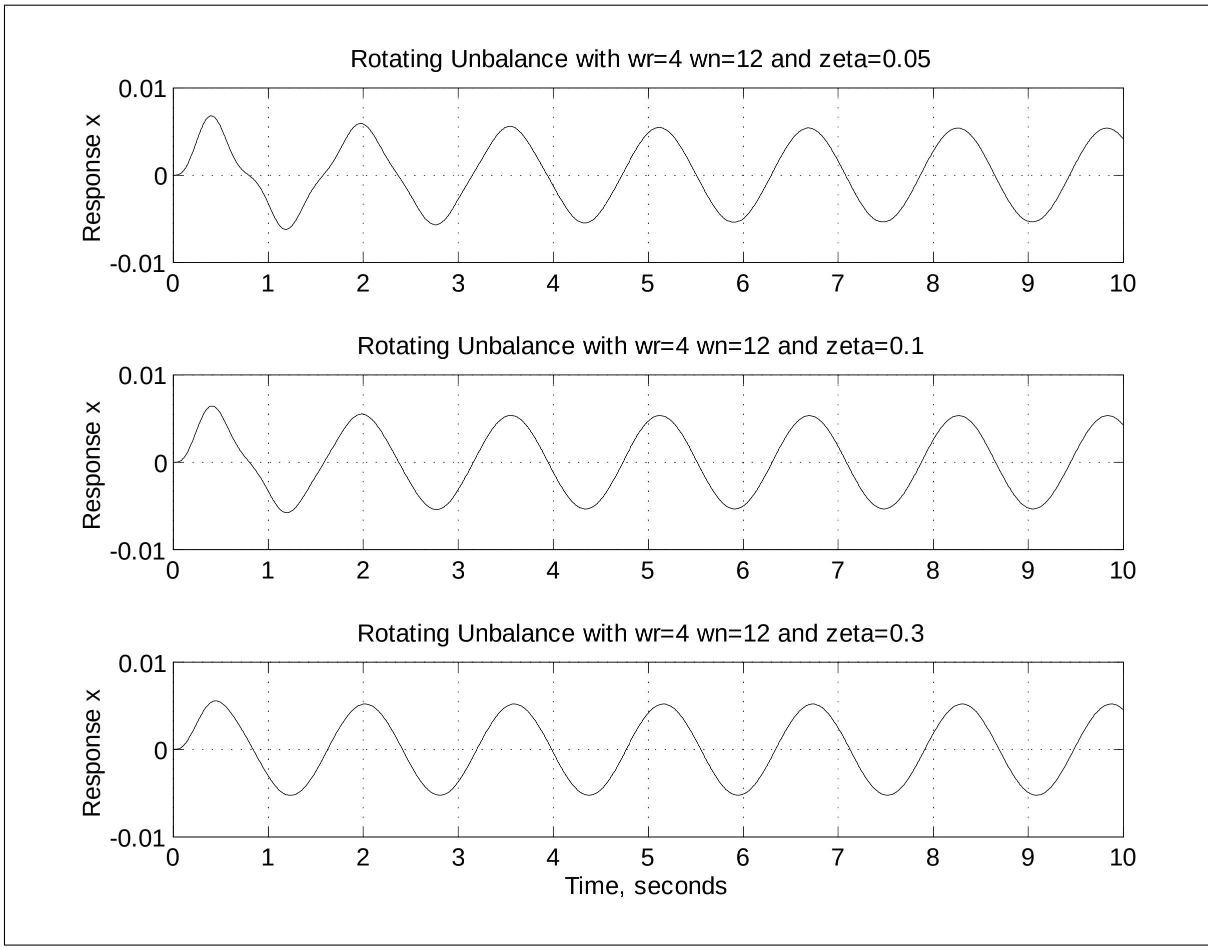

Figure 8.3: Response with varied damping ratios.

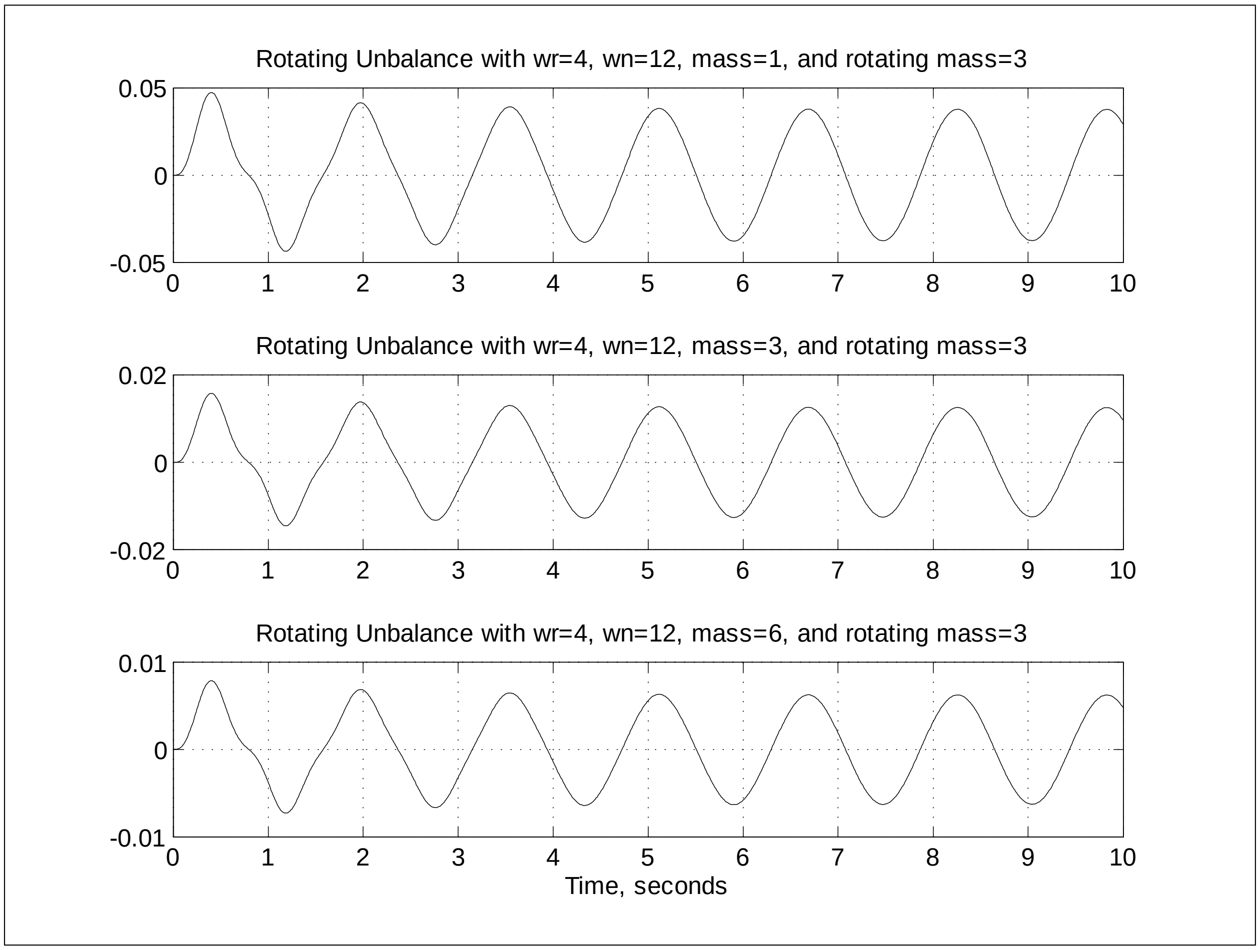

Figure 8.4: Effects of varying system mass.

Chapter 9

Impulse Response of SDOF Systems

In the previous few examples, we have discussed the response of single degree of freedom systems to different forms of sinusoidal inputs. In the following examples, we will examine the effects of non-sinusoidal inputs on single degree of freedom systems. The simplest of these is the impulse response.

An impulse is a force which is applied over a very short time when compared to the period of vibration. The period over which the impulse is applied is assumed to be 2ϵ. If the impulse is centered about time t, then it is applied from $t-\epsilon$ to $t+\epsilon$. If the force has a total value of F_o, then the average value is $F_o/2\epsilon$. So, for all time except the interval around t, the value of the impulse is 0; within the interval, it is $F_o/2\epsilon$.

Using the definition of impulse as force multiplied by time, and noting that impulse is the change in momentum, we see that:

$$(F_o/2\epsilon)\cdot 2\epsilon = mv_o, \tag{9.1}$$

where v_o is the initial velocity of the system. This is because the velocity of the system before the impulse is zero, and it is v_o after the impulse. So, this problem reduces to a single degree of freedom free vibration with zero initial displacement and initial velocity equal to F_o/m. Recall that for a damped oscillator, the response is of the form:

$$x(t) = Ae^{-\zeta\omega t}\sin(\omega_d t+\phi). \tag{9.2}$$

Since

$$A = \sqrt{(v_o+\zeta\omega x_o)^2+(x_o\omega_d)^2}/\omega_d,$$
$$\phi = \tan^{-1}\left(x_o\omega_d/\left(v_o+\zeta\omega x_o\right)\right),$$

we note that, for $x_o = 0$, $A = v_o/\omega_d$, and $\phi = 0$. Thus, Equation 9.2 becomes

$$x(t) = \frac{F_o}{\omega_d} e^{-\zeta\omega t} \sin(\omega_d t). \tag{9.3}$$

This response, as noted above, is simply a single degree of freedom oscillator subject to an initial velocity. The behavior with changing F_o is similar to changing v_o in a single degree of freedom oscillator. For that reason, a program and plots for this situation are not included; they would be the same as those in Example 2. Included in this example is a program which simulates the Dirac delta function (an impulse with $F_o = 1$), for an impulse around a given time and with a given total time interval. The program simply takes the entire interval and sets the function equal to zero for all times other than the one specified, and equal to one for the specified time.

Program 8-2 gives an example of how this delta function can be used inside of another program, by plotting the function over a specified time interval. This program makes use of subfunctions as well. This is more handy for single-use functions, such as several of the unwieldy constants we derived for some of our earlier examples. A subfunction is called just like any other MATLAB function. The subfunction is written after the main code, and is internal to the code. The variables used inside the subfunction stay in the subfunction; they are not introduced to the MATLAB workspace. Thus, if we tried to use the variable "int" in the main code of Program 8-2, we would generate an error. One more item of note is the "\delta" in the ylabel function call. That is a LaTEX tag, which is usable inside of a MATLAB string. Notice the results when the program is run; the y-axis reads "$\delta\left(t_d\right)$," substituting the value entered for t_d. Superscripts, subscripts, Greek characters, and other useful effects are available in this manner; "help latex" at the MATLAB prompt can give you a start on using these.

Program 8-1: delta.m

```
function delta=delta(td,tf)
%This function simulates the delta function.
%The user must input the time at which the
%nonzero value is desired, td, and the ending time, tf.
%
int=tf*td;
%The variable int ensures that a value in the time vector
%will match the desired time. The strategy is to subdivide
%the interval into at least 500 steps (this ensures the
%interval over which the function is nonzero is small).
%
while int<500
int=int*10;
end
t=0:tf/int:tf;
for i=1:int
if t(i)==td
delta(i)=1;
else
delta(i)=0;
end
end
```

```
%Program 8-2: subdemo.m
function [t,y]=subdemo(td,tf)
%
% A simple routine to demonstrate use of subfunctions,
% making use of the delta function as an example.
% Inputs are the time of the delta function, td,
% and the overall timespan, tf.
% Note that subfunctions cannot be used
% inside of script m-files. If we were to
% write Program 8-2 as a script (like most of
% our m-files to this point), the delta function
% would have to be external.
%
y=subdelta(td,tf);
% Calling the subfunction version "subdelta"
% guarantees we'll be calling the subfunction,
% not the function from Program 8-1.
% Now that we have the value of the delta function,
% we need to create a time vector that goes from
% zero to tf, and has the same number of points
% as the vector y (else we will get a plotting
% error). For this, we use the "max" and
% "size" commands.
len=max(size(y));
% "size" returns the vector [nrows ncols], corresponding
% to the number of rows and columns in y. Taking the
% maximum will return the longer dimension, and is a
% shortcut usable for both row and column vectors.
t=linspace(0,tf,len);
% "linspace" is much more natural a command for this
% instance than the colon operator. This will generate
% a vector with len equally-spaced elements between
% 0 and tf.
plot(t,y)
title('Delta Function Sample');
ylabel(['\delta(',num2str(td),')']);
xlabel('Time, seconds');
grid
function delta=subdelta(td,tf)
% This function is Program 8-1.
% Note that a subfunction is included
% with a "function" call, just like an
% external function.
int=tf*td;
while int<500
 int=int*10;
```

```
end
t=0:tf/int:tf;
for i=1:int
 if t(i)==td
 delta(i)=1;
 else
 delta(i)=0;
 end
end
```

Chapter 10

Step Response of a SDOF System

In this example, the force is assumed to be applied instantaneously, but it will be sustained out to infinity. This is essentially an *on-function.* If a force of this sort is plotted versus time, the force looks like a step up. So then, the behavior of the system under this type of load is considered the step response of the single degree of freedom system. We assume the system is underdamped and will have zero initial conditions. We already know the equation of motion,

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = F(t)/m, \tag{10.1}$$

where

$$F(t) = \begin{cases} 0 & \text{if } \ 0 < t < t_o \\ F_o & \text{if } t \geq t_o \end{cases} . \tag{10.2}$$

In order to solve the differential equation, we will use the convolution integral

$$x(t) = \int_0^t F(\tau)g(t-\tau)d\tau. \tag{10.3}$$

Recall that the convolution integral is derived by treating the force as an infinite series of impulse forces. The first fundamental theorem of calculus then allows the infinite series to be treated as the integral given above. The impulse response, just discussed, given by

$$x(t) = \frac{F_o}{m\omega_d}e^{-\zeta\omega_n t}\sin\omega_d t = F_o g(t), \tag{10.4}$$

where we know that

$$g(t) = \frac{1}{m\omega_d} e^{-\zeta\omega t} \sin \omega_d t. \tag{10.5}$$

Therefore,

$$x(t) = \frac{1}{m\omega_d} e^{-\zeta\omega_n t} \int_0^t F(\tau) e^{\zeta\omega_n \tau} \sin \omega_d (t-\tau) d\tau. \tag{10.6}$$

Now, since we have a general expression for $x(t)$, we can substitute our $F(t)$ into Equation 10.6 to give:

$$x(t) = \frac{1}{m\omega_d} e^{-\zeta\omega_n t} \left\{ \int_0^{t_o} (0) e^{\zeta\omega_n \tau} \sin \omega_d (t-\tau) d\tau + \int_{t_o}^t F_o e^{\zeta\omega_n \tau} \sin \omega_d (t-\tau) d\tau \right\}. \tag{10.7}$$

Note that the first term inside the braces is zero. For that reason, for $t < t_o$, the response of the system is zero. To find the response for all other times, we must evaluate the second integral (by parts):

$$x(t) = \frac{F_o}{k} \left\{ 1 - \frac{1}{\sqrt{1-\zeta^2}} e^{-\zeta\omega_n (t-t_o)} \cos\left[\omega_d (t-t_o) - \phi\right] \right\}, t \geq t_o, \tag{10.8}$$

where $\phi = \tan^{-1} \frac{\zeta}{\sqrt{1-\zeta^2}}$. It is important to remember that this equation is only valid for the time after the force is applied; the response is zero before application of the force.

Figure 10.1 shows the variation of the response with the force magnitude. As might be expected, the only difference that results from changing the magnitude of the external force is that the magnitude of the response changes. That is, the magnitude of the response is directly proportional to the magnitude of the external force. The magnitude of the external force also causes a second difference: note that when the oscillatory motion begins, it is not centered around zero. Instead, the mass oscillates around a displacement greater than zero. The value of this center point is also dependent on the magnitude of the external force (from the term 1 in the Equation 10.8).

In Figure 10.2, we vary the natural frequency. This causes two changes in the response. First, the rate of exponential decrease in the response (the effect of damping) is increased; that is, the response stabilizes more quickly. Second, the oscillation frequency decreases, since the natural frequency also dictates the damped frequency.

Finally, Figure 10.3 shows the changes caused by changing the damping ratio. With increasing damping ratio, the amount of time to damp out all vibration decreases. For the third ratio tested, $\zeta = 0.3$, the damping is sufficient to allow no oscillation around the new center point ($x = 1.5$). A second result, which is not immediately evident from the figure but follows from the mathematics, is that the phase angle changes with the damping ratio (note that ϕ is a function of only ζ.)

The three programs used for this example follow below. The way in which the force array is created in Program 9-1 illustrates the convenience afforded the programmer by MATLAB. In Fortran or C programming, we would be forced to use a loop to compare whether the time at each point of the time vector exceeded the step time, and then adjust the force vector appropriately. In MATLAB, the greater-than operator applied to a vector will produce a vector with zeros where the relation is false, and ones where it is true. So if we create a matrix having the results of $t > t_o$ in each row, and array-multiply this matrix by the three-by-*npts* force matrix, we will produce a matrix having step forcing in each row of the appropriate magnitude, activated at $t = t_o$.

The reader may then wonder if the loop calculating the response could have been removed in a similar manner. The answer is "yes," and the programming is left to the reader. The *for* loop is one of the more time-consuming constructions inside MATLAB; thus, for more complicated calculations, it is advantageous to remove these loops from the code; this process is called *vectorization* of code. Using the calculation below as an example, note first that the parameters *phi* and *wd* are calculated outside the loop. These quantities are constant for all forces. Thus, the time-dependent part of all three responses is independent of the forcing parameters. We could then create a matrix that has the time response in each row, and use array multiplication to include the force-dependent parameters. (This is the algorithm for the reader exercise described above.)

Program 9-1: stepfm.m

```
%This program calculates the step response
%of a single degree of freedom system. This
%version tests three different force magnitudes.
%The program assumes a system mass of 1.
%
zeta=0.05;
tf=10;
npts=1000;
t=linspace(0,tf,npts);
to=2;
wn=12;
k=(wn)^2;
for i=1:3
    Fm(i,1)=input('Enter a force magnitude. '); % Forcing Fm to be a
column vector.
end
Fint=Fm*ones(1,npts); % Fint is thus 3-by-npts.
%
% Now, we perform the logical operation t>to,
% and make a 3-by-npts matrix of the result.
%
qtest=t>to;
fmult=[qtest;qtest;qtest]; % Three rows of qtest.
Fo=Fint.*fmult; % The force matrix.
wd=wn*sqrt(1-zeta^2);
phi=atan2(zeta,sqrt(1-zeta^2));
for n=1:3
    A=Fo(n,:)/k;
    B=Fo(n,:)/(k*sqrt(1-zeta^2));
    x(n,:)=A-B.*exp(-zeta*wn*t).*cos(wd*t-phi);
end
for l=1:3
    subplot(3,1,l)
    plot(t,x(l,:))
    title(['Response for wn=',num2str(wn),', Fmax=', ...
    num2str(Fm(l)),', and time=', num2str(to)]);
    ylabel('Response x')
    grid
end
xlabel('Time, seconds')
```

Program 9-2: stepwn.m

```
%This program calculates the step response
%of a single degree of freedom system. This
%version tests three different natural frequencies.
%The program assumes a system mass of 1.
%
Fm=5;
npts=1000;
zeta=0.05;
tf=10;
t=linspace(0,tf,npts);
to=2;
qtest=t>to; % Only one force vector here, so we can do this here.
Fo=Fm*qtest; % Scalar Fm * 1-by-npts vector.
for i=1:3
    wn(i)=input('Enter a natural frequency. ');
    k(i)=wn(i)^2;
end
%
% This could also be vectorized.
%
for n=1:3
    wd=wn(n)*sqrt(1-zeta^2);
    A=Fo/k(n);
    B=Fo/(k(n)*sqrt(1-zeta^2));
    phi=atan2(zeta,sqrt(1-zeta^2));
    x(n,:)=A-B.*exp(-zeta*wn(n)*t).*cos(wd*t-phi);
end
for l=1:3
    subplot(3,1,l)
    plot(t,x(l,:))
    title(['Response for wn=',num2str(wn(l)),', Fmax=', ...
        num2str(Fm),', and time=', num2str(to)]);
    ylabel('Response x')
    grid
end
xlabel('Time, seconds')
```

Program 9-3: stepzeta.m

```
%This program calculates the step response
%of a single degree of freedom system. This
%version tests three different natural frequencies.
%The program assumes a system mass of 1.
%
npts=1000;
Fm=5;
tf=10;
t=linspace(0,tf,npts);
to=2;
qtest=t>to;
Fo=Fm*qtest;
wn=12;
k=(wn)^2;
for i=1:3
    zeta(i)=input('Enter a damping ratio (zeta). ');
end
%
% This could also be vectorized.
%
for n=1:3
    wd=wn*sqrt(1-zeta(n)^2);
    A=Fo/k;
    B=Fo/(k*sqrt(1-zeta(n)^2));
    phi=atan2(zeta(n),sqrt(1-zeta(n)^2));
    x(n,:)=A-B.*exp(-zeta(n)*wn*t).*cos(wd*t-phi);
end
for l=1:3
    subplot(3,1,l)
    plot(t,x(l,:))
    title(['Response for wn=',num2str(wn),', zeta=', ...
        num2str(zeta(l)),', and time=', num2str(to)]);
    ylabel('Response x')
    grid
end
xlabel('Time, seconds')
```

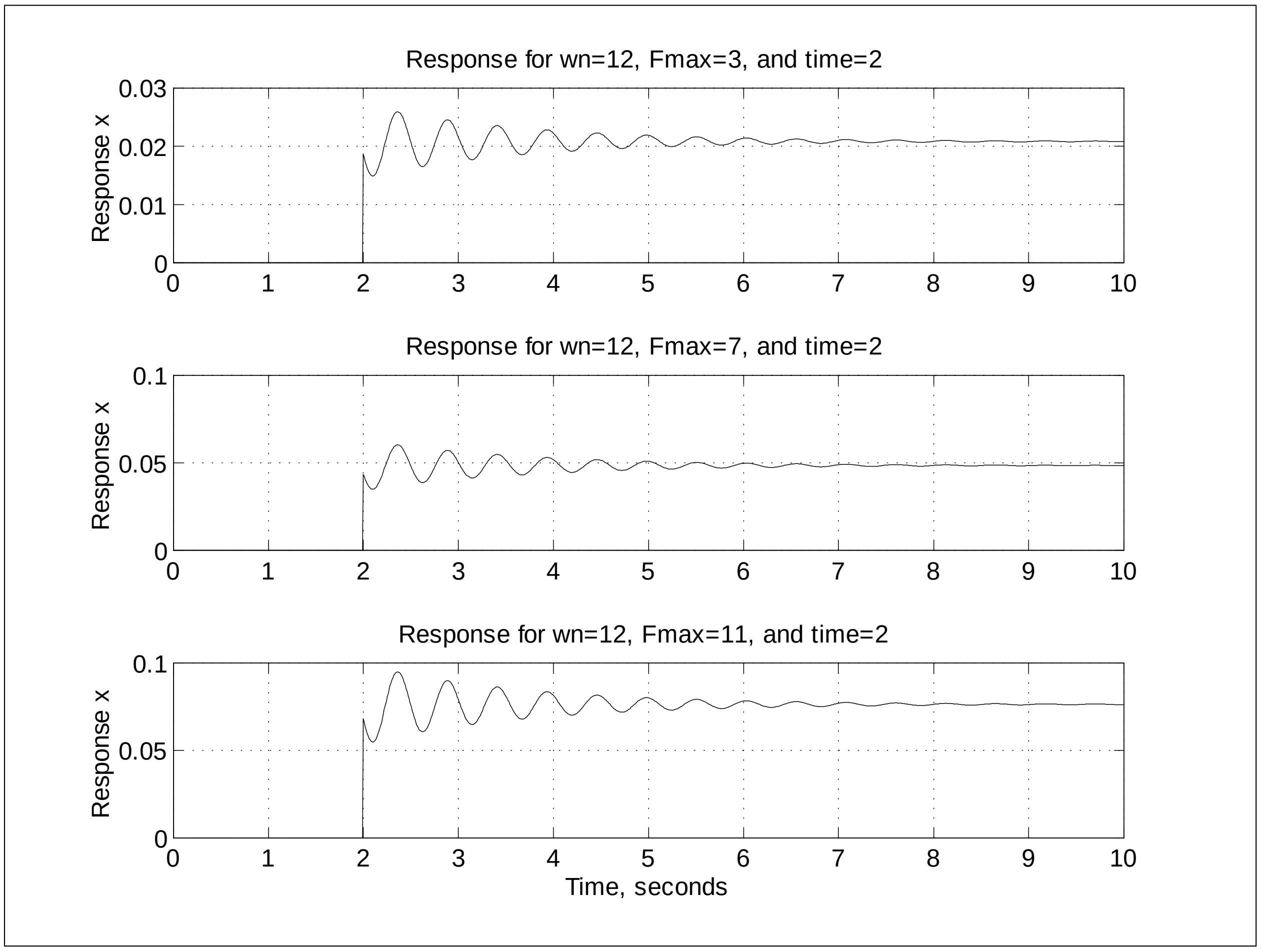

Figure 10.1: Step response of a single degree of freedom system to different step magnitudes.

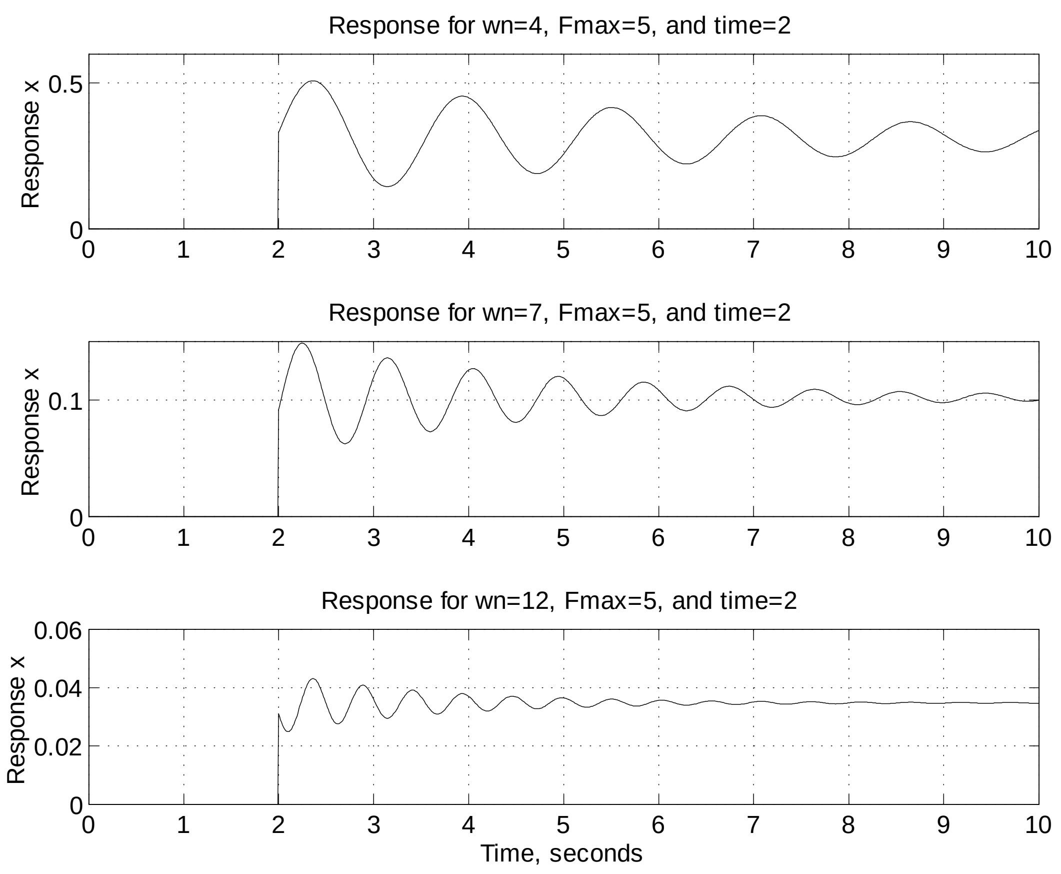

Figure 10.2: Step response of single degree of freedom systems having different natural frequencies.

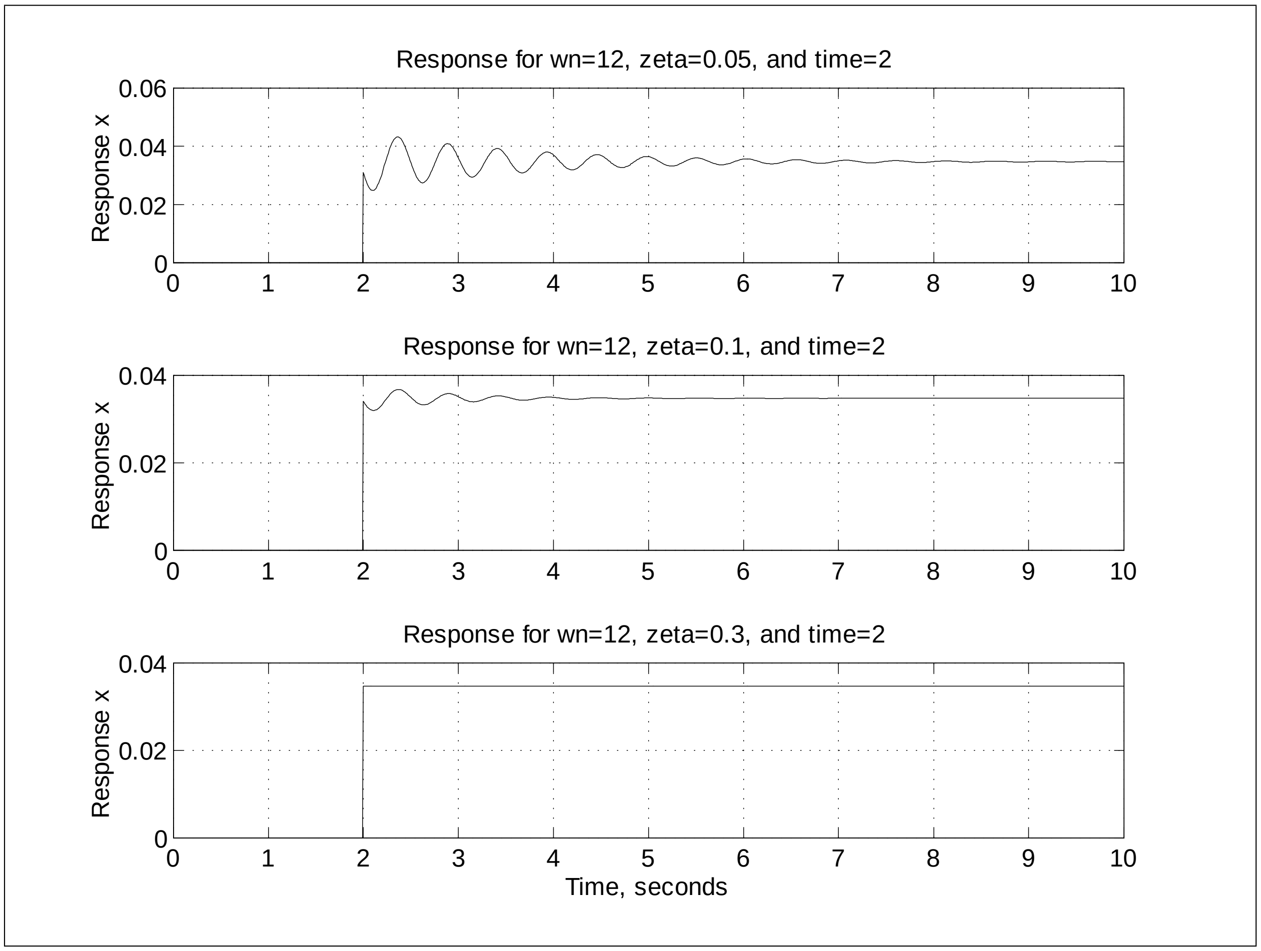

Figure 10.3: Step response of a single degree of freedom system to different levels of damping.

Chapter 11

Response of SDOF Systems to Square Pulse Inputs

A square pulse is a single pulse of constant magnitude and finite duration. To analyze the response of systems to a square wave input, we will treat the square wave as the sum of two equal and opposite step inputs applied at different times. The time interval between application of the step inputs is the duration of the square wave.

Let us assume that the magnitude of the square wave is F_o, and its duration is t_1 seconds. To simulate the wave using step inputs, we begin with a step input of magnitude F_o from time $t = 0$, and add to it at time t_1 a step input of magnitude $-F_{o.}$ By superposition, the total response is the sum of the response of the system to each step input.

Recall that the response of a single degree of freedom system to a step input of magnitude F_m applied at time t_o is:

$$x(t) = \frac{F_m}{k}\left\{1 - \frac{1}{\sqrt{1-\zeta^2}}e^{-\zeta\omega_n(t-t_o)}\cos\left[\omega_d\left(t-t_o\right)-\phi\right]\right\}, t \geq t_o, \tag{11.1}$$

where $\phi = \arctan \zeta/\sqrt{1-\zeta^2}$. If we now consider the two step inputs separately, denoting the response of the system to the input at time $t = 0$ as $x_1(t)$ and the response to the input at time $t = t_1$ as $x_2(t)$, we find that:

$$x_1(t) = \frac{F_o}{k}\left\{1 - \frac{1}{\sqrt{1-\zeta^2}}e^{-\zeta\omega t}\cos\left[\omega_d t-\phi\right]\right\},\ t \geq 0 \tag{11.2}$$

and

$$x_2(t) = -\frac{Fo}{k}\left\{1 - \frac{1}{\sqrt{1-\zeta^2}}e^{-\zeta\omega(t-t_1)}\cos\left[\omega_d\left(t-t_1\right)-\phi\right]\right\},\ t \geq t_1. \quad (11.3)$$

The total response is then:

$$x(t) = \frac{F_o}{k}\left\{1 - \frac{1}{\sqrt{1-\zeta^2}}e^{-\zeta\omega_n t}\cos\left[\omega_d t-\phi\right]\right\},\ 0 \leq t < t_1, \quad (11.4)$$

$$x(t) = \frac{F_o e^{-\zeta\omega_n t}}{k\sqrt{1-\zeta^2}}\left\{e^{\zeta\omega_n t_1}\cos\left[\omega_d\left(t-t_1\right)-\phi\right]-\cos\left(\omega_d t-\phi\right)\right\},\ t \geq t_1. \quad (11.5)$$

Notice how, for the time interval after t_1, the response no longer includes a "$1 -$ " term; the addition of the two responses has removed this term entirely. Again recalling a previous example, the "$1 -$ " term caused the oscillation to be about a new equilibrium (i.e., $x = F_o/k$). Now, since the term has disappeared, the oscillation is centered around zero.

The movement of the center point of the oscillation is best shown in Figure 11.1, which tests three different values of F_o. The oscillation begins about a center point at $x = F_o/k$. When the square wave ends, or, when the equal and opposite step is added, the center point returns to zero. Assume, for a moment, that the magnitude of the second step is not equal to that of the first; call it F_1. In this case, the center point of the oscillation after adding the second step input would be at $x = \left(F_o - F_1\right)/k$. To prove this to yourself, perform the superposition used to obtain Equation 11.5, above (remember that F_1 is negative!). A constant term will remain, and this term will be equal to $\left(F_o - F_1\right)/k$. Another result which is evident from Figure 10.1 is that the change in F_o causes the magnitude of the oscillations to increase, as would be expected from Equation 11.5.

Figure 11.2 shows the effects of changing the natural frequency. Notice that a transition point occurs when the second step input is added. The sudden shift in vibration characteristics is expected, since we have a piecewise expression for $x(t)$. But note that while the transition becomes more abrupt as the natural frequency increases, it is never discontinuous. Since the motion of the mass remains continuous, we can infer that the approach is correct; if we had obtained a discontinuity in the motion, we would know the expression is incorrect. This is because a discontinuous expression would imply that the mass moved from one point to another nonadjacent point without passing through the points in between, a physically impossible situation.

Finally, Figure 11.3 demonstrates the response behavior for three different damping ratios. Again, notice how the high damping ratio ($\zeta = 0.3$) causes all of the vibration to be damped out quickly, so that the mass is practically at rest when the second step input is applied. Again, we see that the transient period decreases with increasing damping ratio.

The MATLAB code to produce these figures demonstrates the vectorization discussed in the previous example. Note how the logical operation $t < t_o$ is used to create a matrix of zeros and ones, where the ones correspond to times before t_o. Then the MATLAB “not” operator (the tilde, ~) is used to arrive at times after t_o. The reader is encouraged to rewrite the vectorized calculation as a loop (in any of the three codes) and see for himself the difference in time expenditure as the number of points is increased by a few orders of magnitude. While there are no fewer actual calculations performed in the vectorized code, the difference is that MATLAB’s vector/matrix calculations are highly optimized, and so the code is faster when handling one matrix calculation instead of many scalar calculations.

One more difference between these codes and the previous is that we’ve inserted another use of the colon operator. As explained in Program 10-1, the statement $a = b(:)$; has a particular meaning in MATLAB. For a vector b, MATLAB will return a column vector a that is equal to a column vector b or the transpose of a row vector b. That is, if we wish to be certain whether a vector we are working with is a column vector, then we can use the colon operator in this manner to force it to be so; the output of $a = b(:)$ is a column vector for *any* vector b.

Another interesting consequence of this use of the colon operator is the result for a *matrix* b. The colon operator would return a column vector a consisting of the rows of b stacked on top of each other. The reader is encouraged to try this, and see the results. Additionally, what would we do if we wanted to force a to be a row vector? Finally, what if we wished to make these script instead functions, taking the number of points in the time vector as inputs? How would we modify the code then? The advantage would be that all the variables from the function would not become part of our workspace. Often, this is a disadvantage from a debugging perspective, but is very nice to have in working code.

```
 % Program 10-1: sqrewn.m
% This program finds the response of a single
% degree of freedom system to a square wave input.
% The wave is assumed to begin at t=0, and lasts until
% t=to. The system mass is again assumed to be equal to 1.
%
npts=1000;
Fm=5;
zeta=0.05;
to=3;
tf=10;
t=linspace(0,tf,npts);
for i=1:3
 wn(i)=input('Enter a natural frequency. ');
 k(i)=sqrt(wn(i));
end
% Demonstrating another use of the colon operator.
% In this sense, wn(:) turns wn into a column
% vector if it was a row vector, and leaves it as
% a column vector if it was one already. Either
% way, we know the result will be a column vector wn.
wn=wn(:);
k=k(:);
%
% To use the logical operators, we need to create a vector
% that is one for t<to, and turn it into a matrix.
%
qtest=t<to;
qpiece=[qtest;qtest;qtest]; % 3-by-npts matrix.
% Now, we need a similar sized matrix for the force magnitude.
Fo=Fm*ones(3,npts); % 3-by-npts, again.
q=sqrt(1-zeta^2);
% In order to calculate the response in a vectorized manner,
% we'll need a wn, k, and wd for each point in time and each
% different value. That can be done by multiplying the
% 3-by-1 matrices we get from our input and calculation
% by ones(1,npts).
%
wd=wn*q; % 3-by-1 matrix.
wnmat=wn*ones(1,npts);
wdmat=wd*ones(1,npts);
kmat=k*ones(1,npts);
phi=atan2(zeta,q);
A=Fm./kmat; % We need a stiffness for each force/time point.
tmat=[t;t;t]; % And a 3-by-npts time matrix.
x1=A.*(1-exp(-zeta*wnmat.*tmat).*cos(wdmat.*tmat-phi)/q);
```

```
x2=A.*exp(-zeta*wnmat.*tmat).*(exp(zeta*wnmat.*to).*cos(wdmat.*(tmat-
to)-phi)-cos(wdmat.*tmat-phi))/q;
x=x1.*qpiece+x2.*(~qpiece); % “not” qpiece.
for l=1:3
 subplot(3,1,l)
 plot(t,x(l,:))
 title(['Square Wave Response for wn=', num2str(wn(l)), ...
', to=',num2str(to),' ,and Fm=', num2str(Fm)])
 ylabel('Response x')
 grid
end
xlabel('Time, seconds')
```

```
%Program 10-2: sqrefm.m
%This program finds the response of a single
%degree of freedom system to a square wave input.
%The wave is assumed to begin at t=0, and lasts until
%t=to. The system mass is again assumed to be equal to 1.
%
npts=1000;
wn=11;
k=(wn)^2;
zeta=0.05;
to=3;
tf=10;
t=linspace(0,tf,npts);
for i=1:3
 Fm(i)=input('Enter a force magnitude. ');
end
Fm=Fm(:); % See above.
% As with the step response example earlier,
% we can use MATLAB's logical operators to
% create a piecewise function without resorting
% to a loop. This can be done with the
% statement: qtest=(t>t1)-(t>t2), when
% the square wave begins at t=t1 and ends at
% t=t2. For the case of t1=0, a simpler
% statement is valid:
qtest=(t<to);
Fo=Fm*ones(1,npts); % Since solution is piecewise, we need Fm throughout.
% This code demonstrates vectorization of this
% piecewise expression. We'll be making use of
% the qtest vector above to tell the code when
% to activate each solution.
%
qpiece=[qtest;qtest;qtest]; % One row for each Fm value.
q=sqrt(1-zeta^2);
wd=wn*q;
phi=atan2(zeta,q);
A=Fo/k; % 3-by-npts matrix.
tmat=[t;t;t]; % Again, 3-by-npts.
x1=A.*(1-exp(-zeta*wn*tmat).*cos(wd*tmat-phi)/q); % Solution for t<to.
x2=A.*exp(-zeta*wn*tmat).*(exp(zeta*wn*to).*cos(wd*(tmat-to)-phi)-cos(wd*tmat-
phi))/q;
x=x1.*qpiece+x2.*(~qpiece); % ~is the "not" operator.
for l=1:3
 subplot(3,1,l)
 plot(t,x(l,:))
 title(['Square Wave Response for wn=', num2str(wn), ...
', to=',num2str(to),' ,and Fm=', num2str(Fm(l))])

 ylabel('Response x')

 grid

end

xlabel('Time, seconds')
```

```
% Program 10-3: sqrez.m
% This program finds the response of a single
% degree of freedom system to a square wave input.
% The wave is assumed to begin at t=0, and lasts until
% t=to. The system mass is again assumed to be equal to 1.
%
npts=1000;
wn=11;
k=(wn)^2;
Fm=7;
to=3;
tf=10;
t=linspace(0,tf,npts);
for i=1:3
 zeta(i)=input('Enter a damping ratio (zeta). ');
end
zeta=zeta(:); % Once more.
% Again we will vectorize the code by using the logical operators
% on t and to. This time it will be somewhat easier, since the
% stiffness and force are scalar. We will need matrices for any
% quantities derived from the damping ratio, and for time.
%
qtest=t<to;
qpiece=ones(3,1)*qtest; % Another way to create the matrix.
tmat=ones(3,1)*t;
zmat=zeta*ones(1,npts); % 3-by-npts.
q=sqrt(1-zmat.^2); % 3-by-npts. Note the array square operator.
wd=wn*q; % wd is 3-by-npts.
phi=atan2(zmat,q); % MATLAB will handle this on an element-by-element
basis.
A=Fm/k; % Scalar.
x1=A*(1-exp(-zmat*wn.*tmat).*cos(wd.*tmat-phi)./q);
x2=A*exp(-zmat*wn.*tmat).*(exp(zmat*wn*to).*cos(wd.*(tmat-to)-phi)-cos(wd.*tmat-
phi))./q;
x=x1.*qpiece+x2.*(~qpiece);
for l=1:3
 subplot(3,1,l)
 plot(t,x(l,:))
 title(['Square Wave Response for wn=', num2str(wn), ...
', to=',num2str(to),' ,and zeta=', num2str(zeta(l))])
 ylabel('Response x')
 grid
end
xlabel('Time, seconds')
```

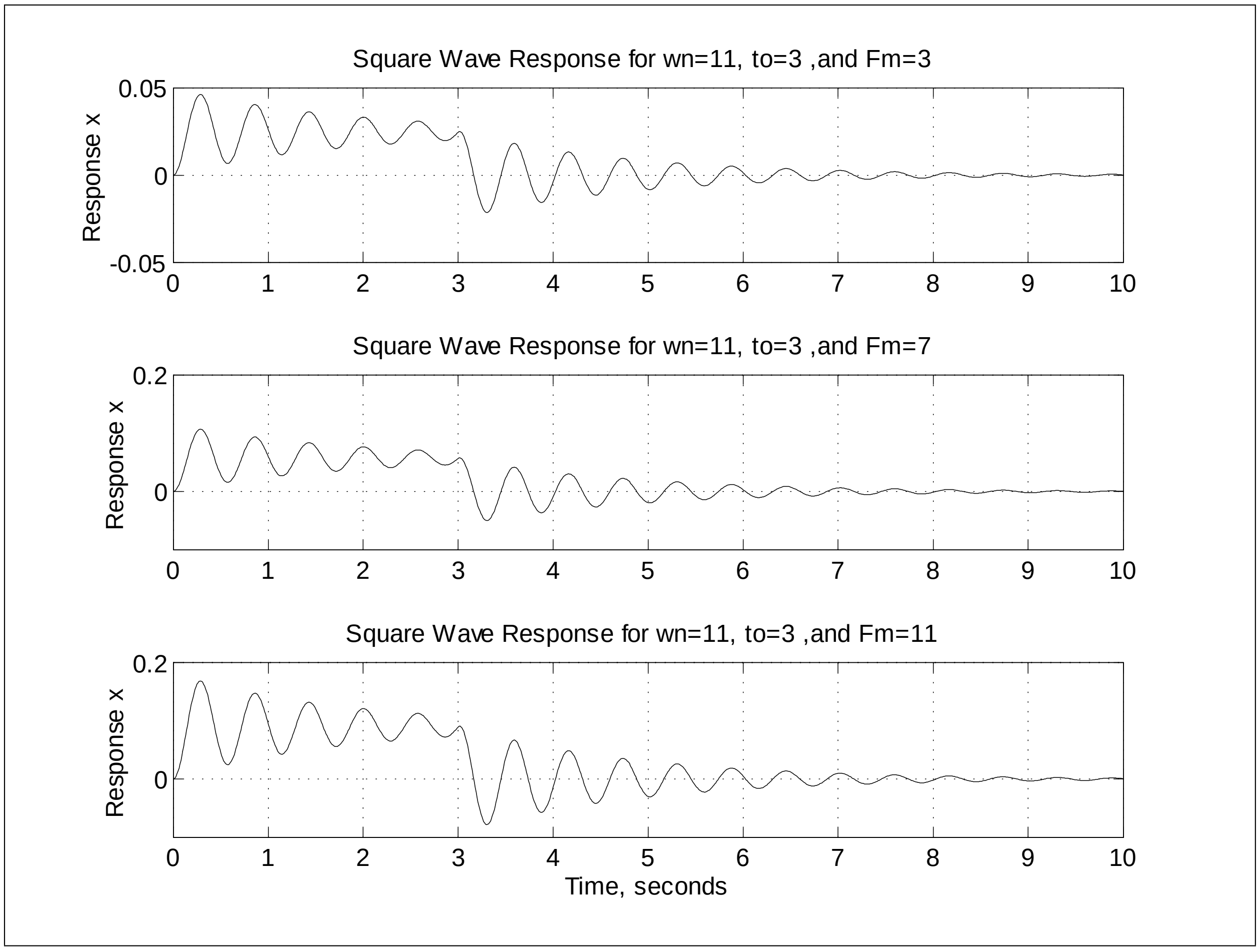

Figure 11.1: Plot of responses for three different force magnitudes.

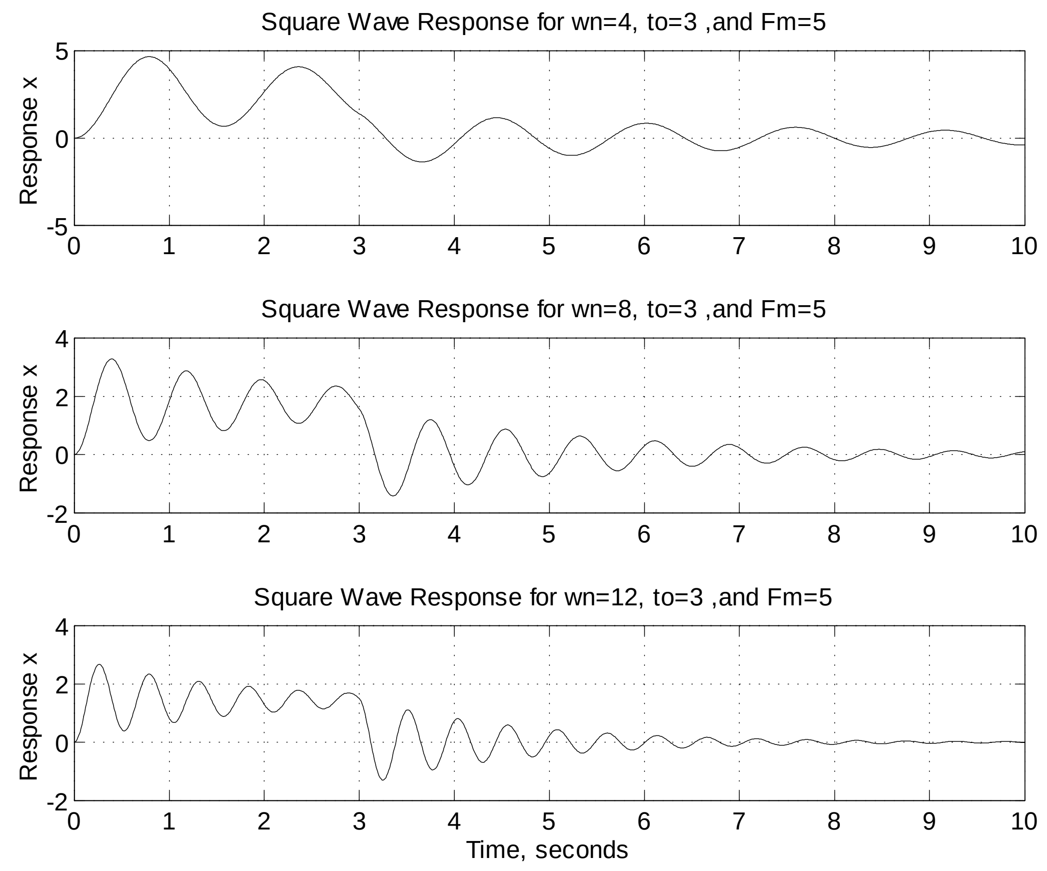

Figure 11.2: Square wave response variation with different natural frequencies.

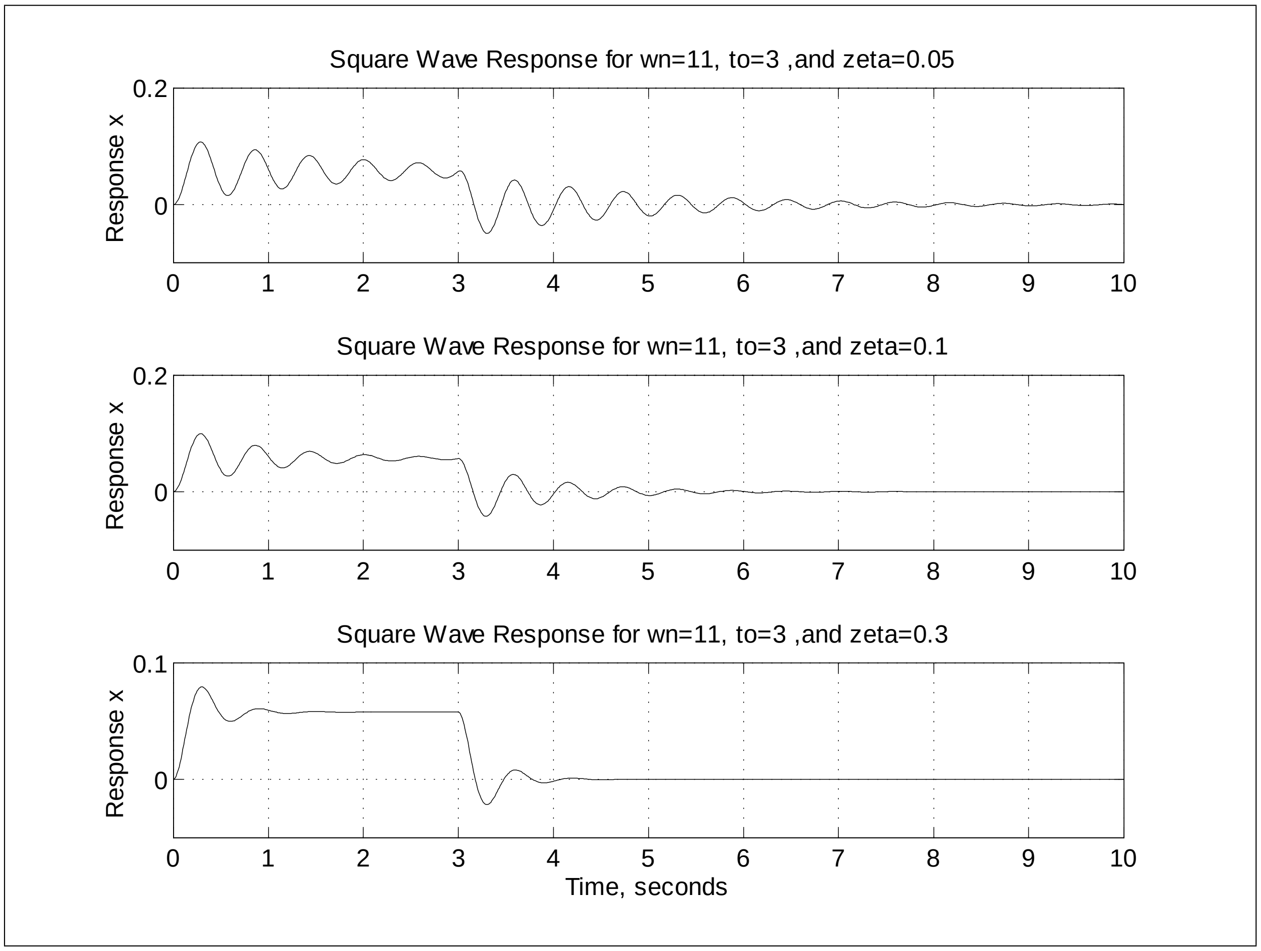

Figure 11.3: Effects of changing damping ratio on the square wave response.

Chapter 12

Response of SDOF System to Ramp Input

Another common form of input encountered in real applications is the ramp input. To examine the response of a single degree of freedom system to this sort of input, we must again apply the convolution integral. Assuming that the load is increased uniformly at a rate of f_o per second and reaches its maximum at time t_d, then the expression for the external force is:

$$F(t) = \begin{cases} f_o t & \text{for } 0 \leq t < t_d, \\ f_o t_d & \text{for } t \geq t_d. \end{cases} \tag{12.1}$$

Substituting the expression for $F(t)$ into the convolution integral yields:

$$x(t) = \begin{cases} \frac{f_o}{m\omega_d} e^{-\zeta\omega_n t} \int_0^t \tau e^{\zeta\omega_n \tau} \sin \omega_d(t-\tau) d\tau & \text{for } 0 \leq t < t_d, \\ \frac{f_o}{m\omega_d} e^{-\zeta\omega_n t} \left[\int_0^{t_d} \tau e^{\zeta\omega_n \tau} \sin \omega_d(t-\tau) d\tau + t_d \int_{t_d}^t e^{\zeta\omega_n \tau} \sin \omega_d(t-\tau) d\tau \right] & \text{for } t \geq t_d. \end{cases} \tag{12.2}$$

This expression is best evaluated using MAPLE or a table of integrals.

$$x(t) = \frac{f_o}{m\omega_d} e^{-\zeta\omega_n t} \left(\frac{1}{\left(\zeta^2\omega_n^2 + \omega_d^2\right)^2} \right) \left\{ \omega_d e^{\zeta\omega_n t} \left[t\zeta^2\omega_n^2 + t\omega_d^2 - 2\zeta\omega_n \right] + 2\zeta\omega_n\omega_d \cos(\omega_d t) \right.$$

$$\left. + \left(\zeta^2\omega_n^2 - \omega_d^2\right) \sin(\omega_d t) \right\}, \; 0 \leq t < t_d,$$

$$x(t) = \frac{f_o}{m\omega_d} e^{-\zeta\omega_n t} \left(\frac{1}{\left(\zeta^2\omega_n^2 + \omega_d^2\right)^2} \right) \left\{ \omega_d e^{\zeta\omega_n t} \left[t\zeta^2\omega_n^2 + t\omega_d^2 - 2\zeta\omega_n \right] + 2\zeta\omega_n\omega_d \cos(\omega_d t) \right.$$

$$\left. + \left(\zeta^2\omega_n^2 - \omega_d^2\right)\sin\left(\omega_d t\right) \right\} + \frac{f_o t_d}{k} - \frac{f_o t_d}{k\sqrt{1-\zeta^2}} e^{-\zeta\omega_n (t-t_d)} \cos(\omega_d(t-t_d)-\phi),\ t \geq t_d. \tag{12.3}$$

Note that k represents the system stiffness. This solution is reflected in the MATLAB code that follows. An important note about the solution for this expression is that there is no equilibrium position, as seen for the step and square wave responses, until after the input has levelled off. This is because the constant that creates the new center point is the result of an integration that does not start at zero, and no such integration exists in this solution until after t_d.

From Figure 12.1, one would infer that the transition to the new equilibrium of vibration is discontinuous, as it was in the step and square wave responses. Notice how the response seems to be nonexistent for the first few seconds, until the load is fully applied, and then begins oscillating, as in the step response. So, at this point, it is reasonable to assume that the ramp response and step response of a single degree of freedom system are similar.

However, Figure 12.2 shows that this is not the case. This figure only shows the response during the transient loading period. Notice how the system is oscillating during this period, around a constantly increasing equilibrium. That is, if a line was drawn through the identical point on each period of the sinusoid, the result would be a line of positive slope. This shows that the ramp response does have a subtle difference from the step response; the ramp response has less deflection at the point in time that the full load is applied than the step response.

The Matlab code below follows the same general structure we have seen previously. We vectorize the code, so that we apply Matlab's strength (vector math) to the problem, avoiding the slower loops. Note that both portions of the piecewise solution for $x(t)$ are multiplied by the same constant factor. Thus, we initialize the variable $a1$ in Program 10-1 once. Also, we use vector multiplication to construct two solution matrices over the entire time interval. Finally, we make use of Matlab's logical operators to assemble the full piecewise solution.

```
 % Program 11-1: rampfo.m
%This program solves the ramp response of a
%single degree of freedom system. The external
%force is assumed of the form F(t)=fo*t, and
%is assumed to last for te seconds, after which
%it levels off at F=fo*te until infinity.
%Again, a unit system mass is assumed.
npts=1000;
wn=5;
zeta=.05;
wd=wn*sqrt(1-zeta^2);
k=wn^2;
te=4;
for kr=1:3
 fo(kr)=input('Enter a force magnitude (fo). ');
end
fo=fo(:); % Force fo to be a column vector.
tf=10;
t=linspace(0,tf,npts);
qtest1=t<te;
qtest2=t>te;
% This time, we need two conditional matrices.
amult1=[qtest1;qtest1;qtest1];
amult2=[qtest2;qtest2;qtest2];
% This constant is used for both parts.
a1=fo/(k*(zeta^2*wn^2+wd^2)^2);
% The solution for t<te.
num1=t*(zeta*wn)^2+t*wd^2-2*zeta*wn;
num2=exp(-zeta*wn*t).*(2*zeta*wd*wn*cos(wd*t)+ ...
 ((zeta*wn)^2-wd^2)*sin(wd*t));
x1=a1*(num1+num2); % (3-by-1)*(1-by-npts)=(3-by-npts).
% Now, the solution for t>te.
num3=te*(zeta*wn)^2+te*wd^2-2*zeta*wn;
num4=exp(-zeta*wn*te)*(2*zeta*wd*wn*cos(wd*te)+ ...
 ((zeta*wn)^2-wd^2)*sin(wd*te));
a=fo*te/k;
q=sqrt(1-zeta^2);
phi=atan2(zeta,q);
r=a/q*(1-exp(-zeta*wn*(t-te)).*cos(wd*(t-te)-phi));
s=a1*(num3+num4)*ones(1,npts); % num3 and num4 are scalars, so we
must vectorize this.
x2=r+s;
x=x1.*amult1+x2.*amult2; % This gives a matrix of three 1-by-npts solution
vectors.
for i=1:3
 subplot(3,1,i)
 plot(t,x(i,:))
 title(['Response for wn=',num2str(wn),', fo=', ...
num2str(fo(i)),' zeta=',num2str(zeta)])
 ylabel('Response x')
 grid
end
xlabel('Time, seconds')
```

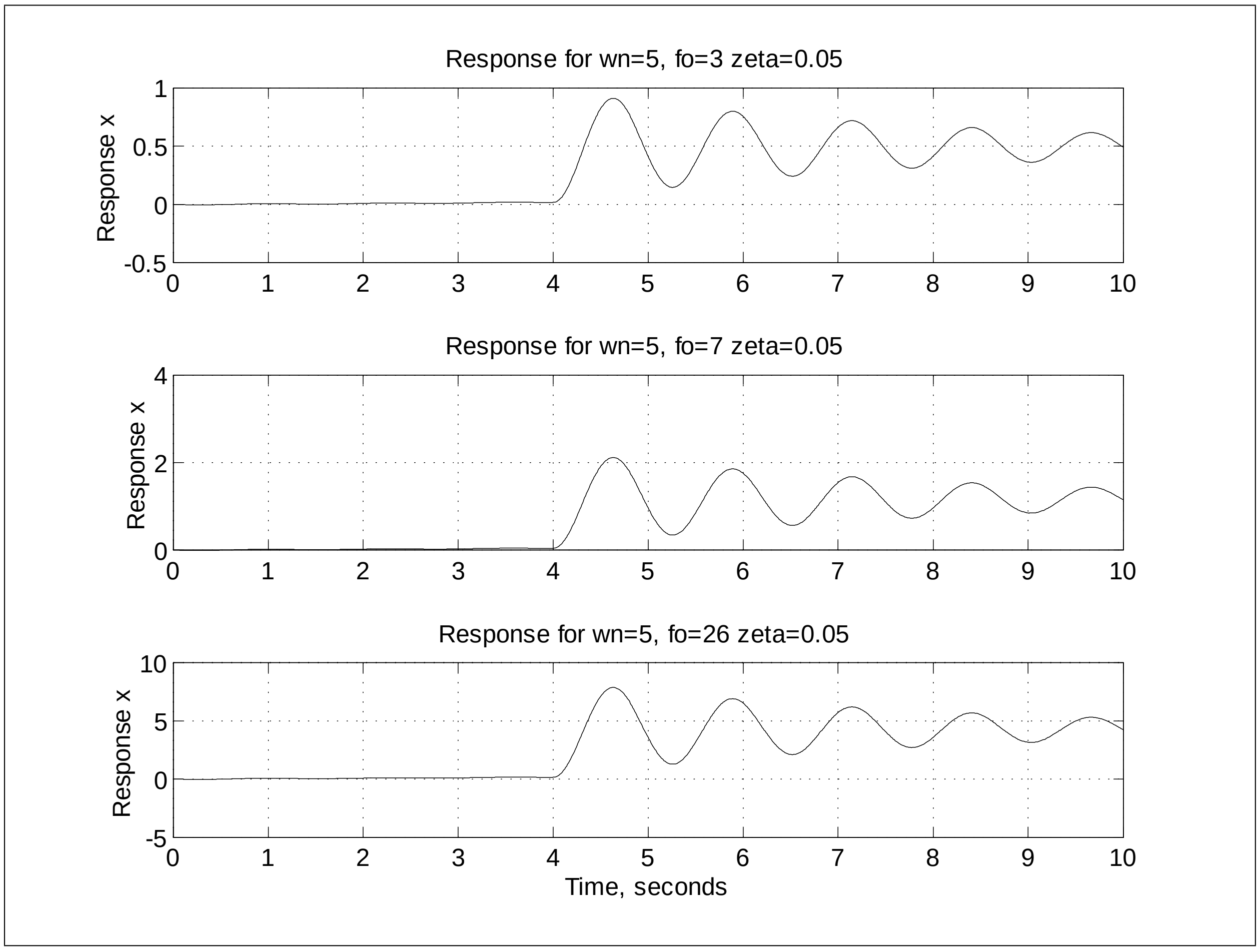

Figure 12.1: Response of a single degree of freedom system to different rates of loading.

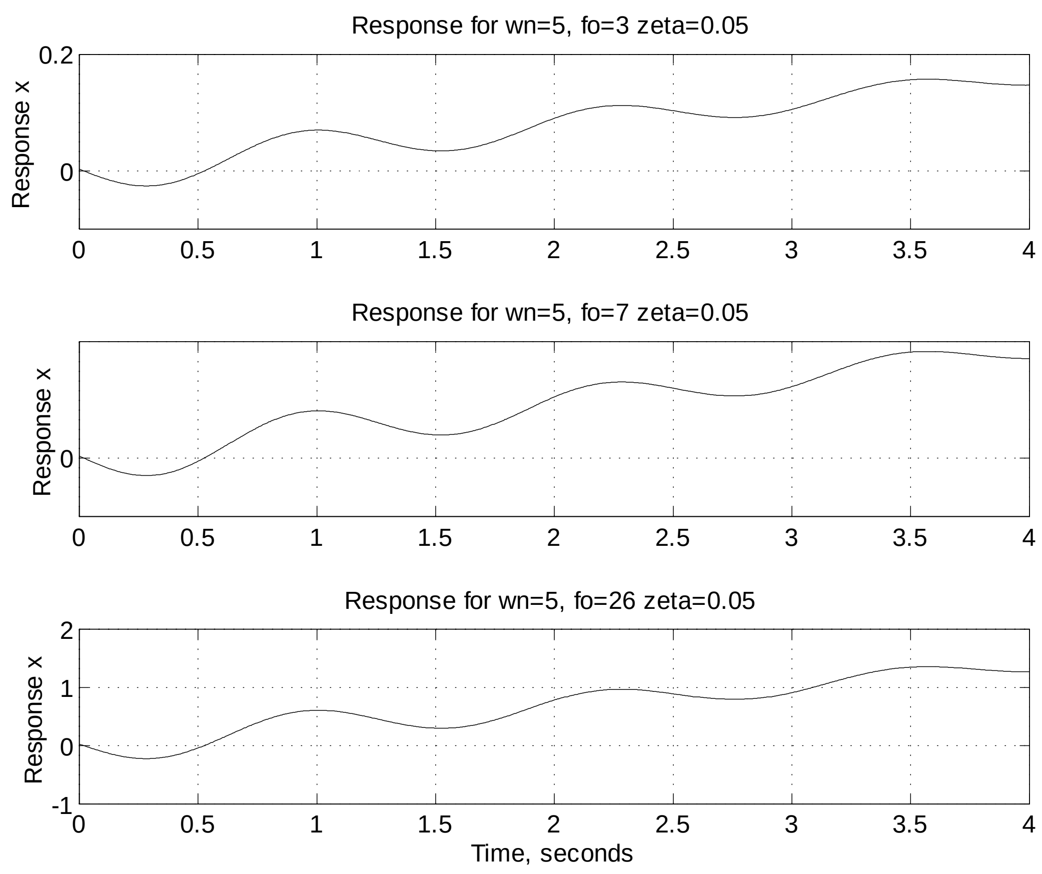

Figure 12.2: Focusing on the first few seconds of oscillation, showing that the system does oscillate during the transient period.

Chapter 13

Response of SDOF Systems to Arbitrary Periodic Input

An arbitrary periodic input can be thought of as any periodic input that cannot be expressed as a single sine or cosine function. For example, a sum of sines and cosines is an arbitrary periodic input. The most straightforward method to analyze the response to a periodic input is the Fourier series. A Fourier series representation of an arbitrary periodic function can take the form:

$$F(t) = \frac{a_o}{2} + \sum_{n=1}^{\infty} \left(a_n \cos n\omega_T t + b_n \sin n\omega_T t \right), \tag{13.1}$$

where T is the period of the function, and $\omega_T = \frac{2\pi}{T}$. The coefficients are obtained by using the orthogonality properties of harmonic functions:

$$a_n = \frac{2}{T} \int_0^T F(t) \cos n\omega_T t dt, \; n = 0, 1, 2, ...$$

$$b_n = \frac{2}{T} \int_0^T F(t) \sin n\omega_T t dt, \; n = 1, 2, ...$$

These coefficients can be easily evaluated using the well-known integrals:

$$\int_0^T \sin n\omega_T t \sin m\omega_T t dt = \begin{cases} 0 & \text{if } m \neq n, \\ T/2 & \text{if } m = n, \end{cases}$$

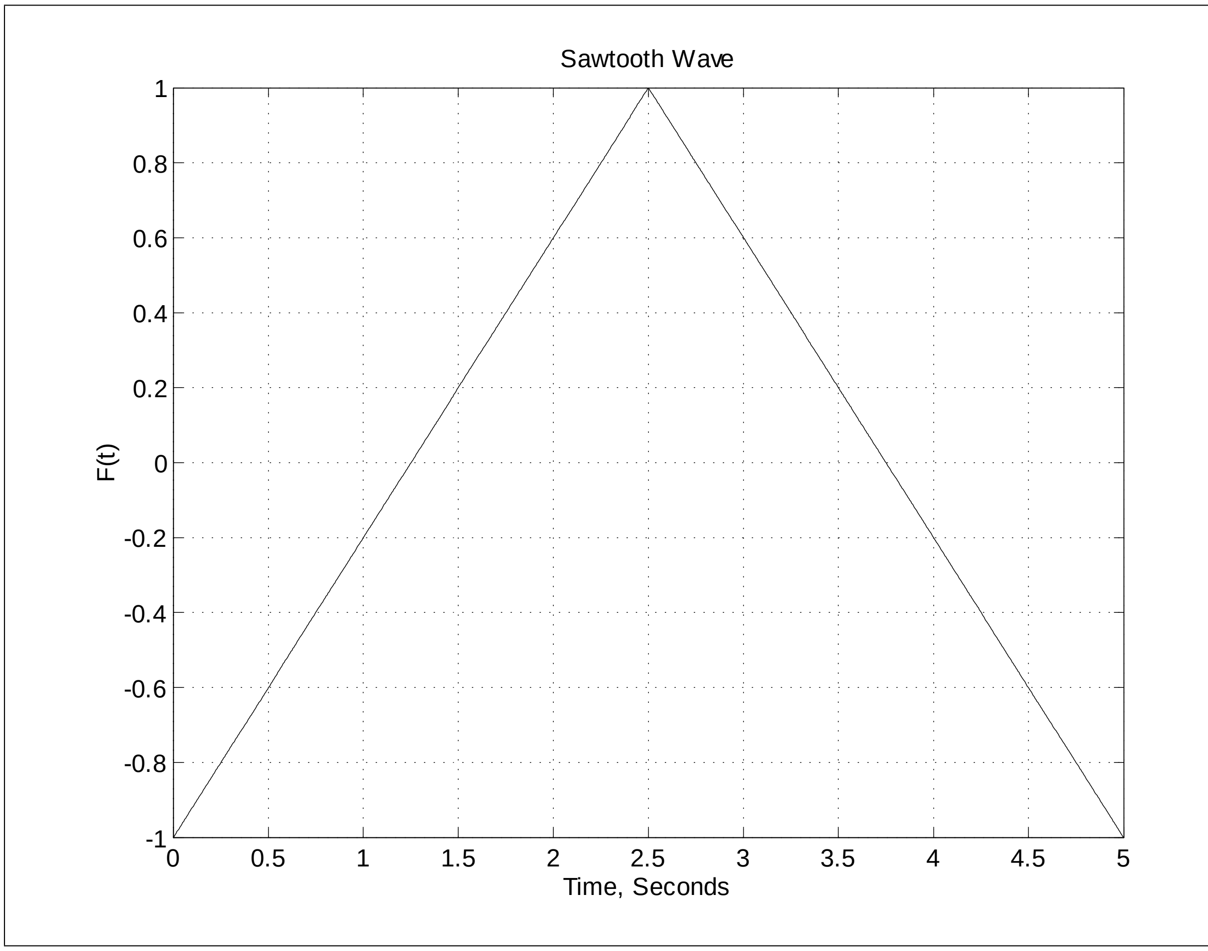

Figure 13.1: Sawtooth wave to be defined by a Fourier series. One period is shown.

$$\int_0^T \cos n\omega_T t \cos m\omega_T t dt = \begin{cases} 0 & \text{if } m \neq n, \\ T/2 & \text{if } m = n, \end{cases}$$

$$\int_0^T \cos n\omega_T t \sin m\omega_T t dt = 0.$$

Using these relationships, the evaluation of the Fourier coefficients will become reasonable. As a demonstration, Figure 13.1 shows a plot of the sawtooth wave for $T = 5$. The wave is defined by:

$$F(t) = \begin{cases} \frac{4}{T}t - 1 & \text{for } 0 \leq t \leq \frac{T}{2}, \\ 1 - \frac{4}{T}\left(t - \frac{T}{2}\right) & \text{for } \frac{T}{2} \leq t \leq T. \end{cases} \tag{13.2}$$

The Fourier coefficients which define this function are $a_0 = 0$,

$$b_n = \frac{2}{T}\left\{\int_0^{T/2}\left(\frac{4}{T}t - 1\right)\sin n\omega_T t\ dt + \int_{T/2}^{T}\left[1 - \frac{4}{T}\left(t - \frac{T}{2}\right)\right]\sin n\omega_T t\ dt\right\}. \tag{13.3}$$

Using Maple, we find that this reduces to:

$$b_n = \frac{4\sin(n\pi) - 2\sin(2n\pi) + n\pi\cos(2n\pi)}{n^2\pi^2} - \frac{1}{n\pi},$$

where n is an integer, $\sin(n\pi) = \sin(2n\pi) = 0$, and $\cos(2n\pi) = 1$. Thus, we obtain:

$$b_n = \frac{n\pi}{n^2\pi^2} - \frac{1}{n\pi} = 0.$$

The only nonzero terms that describe this wave are the a_n terms:

$$a_n = \frac{2}{T}\left\{\int_0^{T/2}\left(\frac{4}{T}t - 1\right)\cos n\omega_T t\ dt + \int_{T/2}^{T}\left[1 - \frac{4}{T}\left(t - \frac{T}{2}\right)\right]\cos n\omega_T t\ dt\right\},$$

or

$$a_n = \frac{4\cos n\pi - 2\cos 2n\pi - n\pi \sin 2n\pi - 2}{n^2\pi^2}. \tag{13.4}$$

Note that since n is an integer, $\sin(2n\pi) = 0$ for all n. For even n, $a_n = 0$. However, for odd n, $a_n = -8/n^2\pi^2$. Therefore,

$$F(t) = -\frac{8}{\pi^2}\left[\cos\frac{2\pi}{T}t + \frac{1}{9}\cos\frac{6\pi}{T}t + \frac{1}{25}\cos\frac{10\pi}{T}t + ...\right]. \tag{13.5}$$

To evaluate this expression using Matlab, we would need to perform two simple steps. First, we would need to find a point where the terms of the series become negligible, to our desired level of accuracy. Then, we would need to solve the vibrating system for each term of the infinite series and superpose these results. This process is not compatible with Matlab; the program needed to perform this would be too long. For this reason, no such program is included here. The program below shows that a few terms of the infinite series can provide a very close approximation to the total series.

This program is written as a function, taking as input the desired number of terms in the Fourier series. The time span is presumed to be 5 seconds. In Figure 13.2, we plot the results for several numbers of terms to show the convergence. To get different sawtooth waves, one would have to either change T (to get a different period) or add a premultiplying factor (to change the amplitude). Such improvements are left to the reader.

```
function [t,F]=foursaw(nterm)
% Program 1: foursaw.m
% This function returns the
% Fourier representation for
% a sawtooth wave having nterm
% number of nonzero Fourier coefficients,
% as well as a corresponding time
% vector (for ease in plotting).
%
t=linspace(0,5,500);
T=5; % Period of 5 seconds.
F=zeros(size(t));
if nterm<1
 error('Number of terms must be a positive integer!')
end
for i=1:nterm
 n=2*i-1; % Getting odd values only.
 F=F+(-8/pi^2)*cos(2*pi*n/T*t)/n^2;
end
```

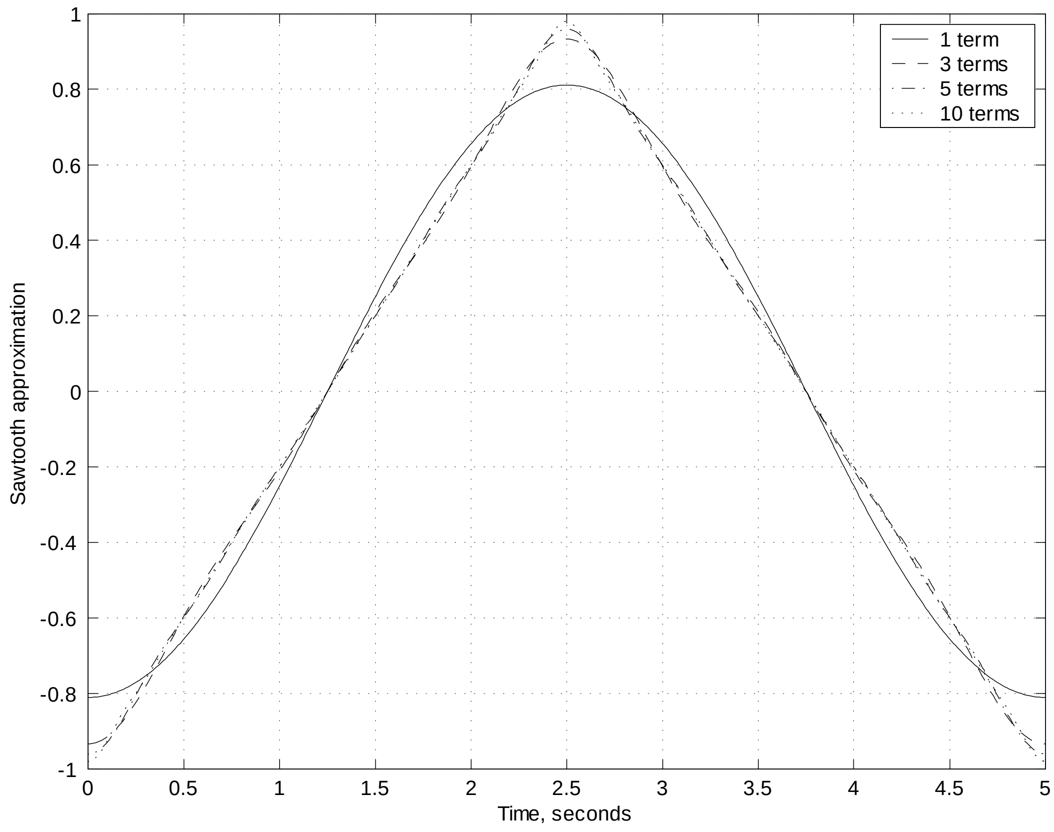

Figure 13.2: Four sawtooth approximations. Note that 10 terms provide a very good approximation to the actual function.

Chapter 14

Free Vibration of MDOF Systems

The simplest multi-degree of freedom system is, of course, the two degree of freedom system, for which we will consider free vibration in this example. Assume that the system is as pictured in Figure 14.1. Note that we need two coordinates to define this system's motion. Applying Newton's second law to each mass results in two governing equations

$$m_1\ddot{x}_1 + (k_1 + k_2)x_1 - k_2x_2 = 0 \tag{14.1}$$

$$m_2\ddot{x}_2 - k_2x_1 + (k_2 + k_3)\,x_2 = 0. \tag{14.2}$$

The motion of the masses are coupled. To solve these equations, it would be advantageous if we could find a coordinate system in which the equations were not coupled. Then, the problem would reduce to two single degree of freedom

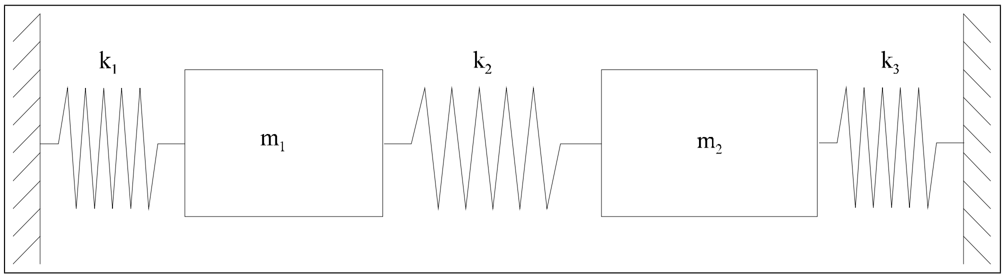

Figure 14.1: Typical two degree of freedom free oscillator.

systems, which we have already solved. The procedure of decoupling can be shown in matrix notation:

$$\begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix} \ddot{\mathbf{x}} + \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 + k_3 \end{bmatrix} \mathbf{x} = \mathbf{0}. \tag{14.3}$$

where $\mathbf{x} = \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix}$, and $\ddot{\mathbf{x}} = \begin{Bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{Bmatrix}$. Let us denote the matrix multiplying $\ddot{\mathbf{x}}$ by M (the mass matrix) and the matrix multiplying $\mathbf{x}$ by K. The modal analysis procedure may then be summarized as follows:

1. Find the eigenvalues and matrix of eigenvectors for the matrix $K - \lambda M$. Let this matrix of eigenvectors be U, and denote its columns (the eigenvectors) by $\{u\}_i$.

2. Normalize the eigenvectors with respect to the mass matrix through the products $\{u\}_i^T M \{u\}_i = 1, \{u\}_i^T K \{u\}_i = \omega_i^2 = \lambda_i$, where ω_i is the natural frequency of mode i.

3. From the solution of the motion in the decoupled modal coordinates, we arrive at:

$$\mathbf{x}(t) = \sum_{i=1}^{2} \{u\}_i \left(\{u\}_i^T M\mathbf{x}(0) \cos \omega_i t + \frac{1}{\omega_i} \{u\}_i^T M\dot{\mathbf{x}}(0) \sin \omega_i t \right) \tag{14.4}$$

The main benefit of this procedure (from a MATLAB standpoint) is that it is easily extended to n degrees of freedom by changing the limit of the summation in Equation 14.4 from 2 to n. Of course, we would also have n natural frequencies and eigenvectors, but the procedures for finding them (i.e., solving the eigenvalue problem) and normalizing the eigenvectors with respect to the mass matrix are unchanged. The following program, *modfree.m*, is capable of solving n degree of freedom free vibration through the modal analysis procedure outlined above.

The following five figures show the results of different masses, stiffnesses, and initial conditions on the motion. Recall that this is free vibration; zero initial conditions will lead to zero motion. For the first three figures, $m_1 = 1$, $m_2 = 4$, $k_1 = k_3 = 10$, and $k_2 = 2$. Since the masses and stiffnesses are not changing, the natural frequencies for these cases will be identical: $\omega_1 = 3.480$, and $\omega_2 = 1.700$. For Figure 14.2, the initial displacement of each degree of freedom is 1. Notice how the second mode of vibration seems to show little evidence of the first mode; it looks like a single sinusoid, instead of a sum of sinusoids. This is due to the difference in mass between the two degrees of freedom; remember that the second mass is four times the first.

As validation for this assertion, look at Figure 14.3. This figure has zero initial velocity as well, but the initial displacements are now 3 and –2, respectively. Again, the second mode is vibrating along a single sinusoid; since the mass ratio is unchanged, the mode behavior is as well. If we eliminated an initial displacement from the second degree of freedom, we would expect the system to vibrate in both modes, at least in the second degree of freedom. As Figure 14.4 shows, this is the case. Notice that the first mode only vibrates in one frequency this time; this is because the displacement of the second mass and its velocity are too small to impart any appreciable force on the first mass.

As a final proof of the effects of the mass on the mode shapes, look at Figure 14.5. This figure was created by interchanging the masses; the mass of 4 units is in the first degree of freedom, and the mass of 1 unit is in the second. Notice how the first mode shape is now a single sinusoid, and the second is a sum. A peculiarity to this case is that the system is symmetric; that is, the response of the first mode in Figure 14.2 matches that of the second mode in Figure 14.5, and vice versa. This is due to the symmetric stiffness matrix. If we change the value of either k_1 or k_3, the symmetric behavior would no longer exist.

Figure 14.6 shows the effects of adding an initial velocity to the system. As expected from a single degree of freedom system, the initial velocity affects the slope of the response curve at the starting point, but has no effect on the qualitative aspects of the mode shapes, or on the natural frequencies. Also, the maximum amplitude of vibration is increased for both modes.

Figure 14.7 shows the result of changing the mass matrix to $m = \begin{bmatrix} 3 & 0 \\ 0 & 9 \end{bmatrix}$. As a result, $\omega_1 = 2.014$, and $\omega_2 = 1.131$. These reduced natural frequencies are reflected in the figure; when compared to Figure 14.2, the curves of Figure 14.7 have a lower frequency. The curves look a little more "spread out" than those in Figure 14.2. However, the second mode shape is still a single sinusoid, not a sum like the first shape. Note that both initial displacements were again 1.

Finally, Figure 14.8 shows the result of changing k_1 and k_3 to 1 unit, and k_2 to 3 units. The natural frequencies become $\omega_1 = 2.150$, and $\omega_2 = 0.6154$. Again, the difference between Figure 14.2 and Figure 14.8 is readily apparent. A peculiarity that develops is that the first mass's vibration comes in more distinct peaks; one peak reaches 1, and the next is about 0.7. In Figure 14.2, the difference is 1 to less than 0.5.

Program 13-1: modfree.m

```
% This code solves n-degree of freedom free
% vibration using the modal analysis techniques
% of Section 7.6.1.
%
clear;
M=input('Enter the mass matrix: ');
[n,o]=size(M);
if n~=o then
error('M matrix must be square!');
end
K=input('Enter the stiffness matrix: ');
[n,o]=size(K);
if n~=o then
error('K matrix must be square!');
end
qu=0;
[u,l]=eig(K,M);
% Using "eig" in this way allows us to subtract M*w^2
% from K, instead of I*w^2 (where I is the n by n identity
% matrix).
% The output from "eig" gives unit-length eigenvectors.
% We need to scale them with respect to M.
%
for s=1:n
alfa=sqrt(u(:,s)'*M*u(:,s));
u(:,s)=u(:,s)/alfa;
end
x0=input('Enter the initial displacement column vector: ');
xd0=input('Enter the initial velocity column vector: ');
tf=input('Enter the final time: ');
t=0:0.1:tf;
q=tf/0.1;
x=zeros(size(n,q));
% Applying Equation 7.183.
%
for j=1:n
w(j)=sqrt(l(j,j));
xt=u(:,j)*(u(:,j)'*M*x0*cos(w(j).*t)+u(:,j)'*M*xd0/...
w(j)*sin(w(j).*t));
x=x+xt;
end
% Plotting the modes in a subplot format.
% Note that, for more than 3 or 4 degrees
% of freedom, the plots will become nearly
% unreadable.
```

```
%
for r=1:n
subplot(n,1,r)
plot(t,x(r,:))
xlabel('Time, seconds');
ylabel(['Response x',num2str(r)]);
end
```

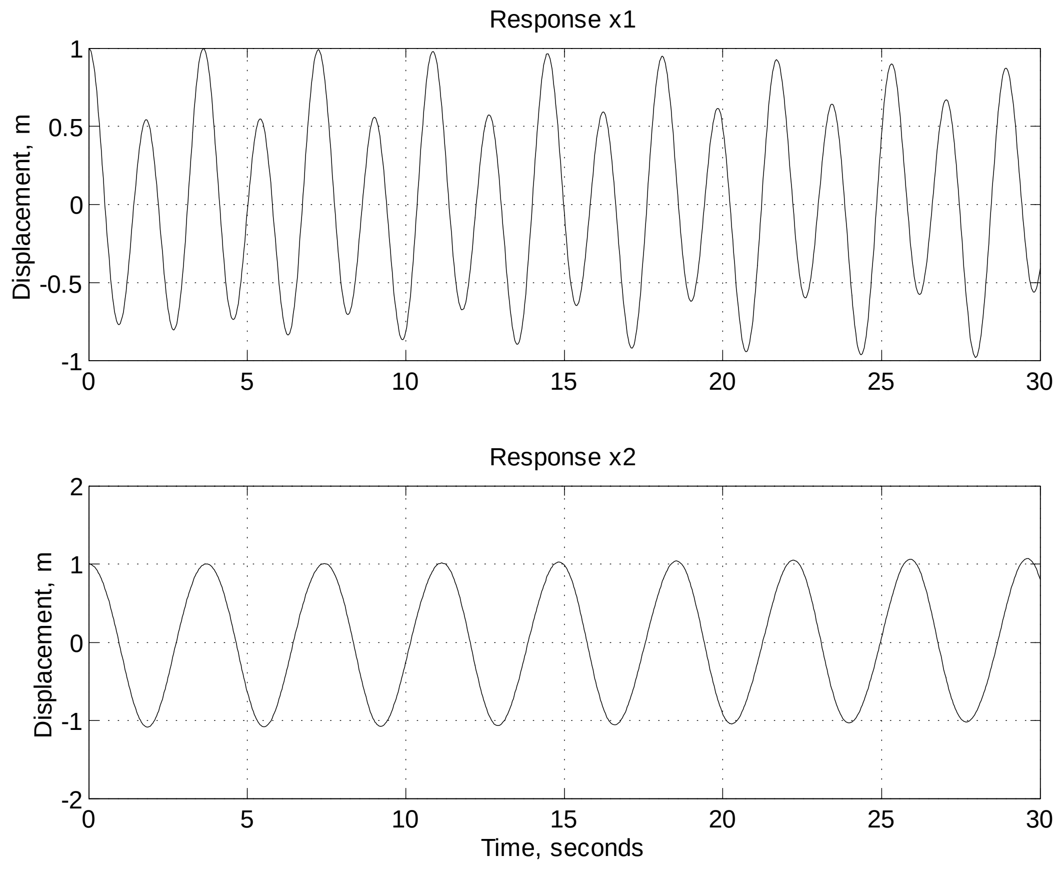

Figure 14.2: Response to initial displacement vector of [1 1].

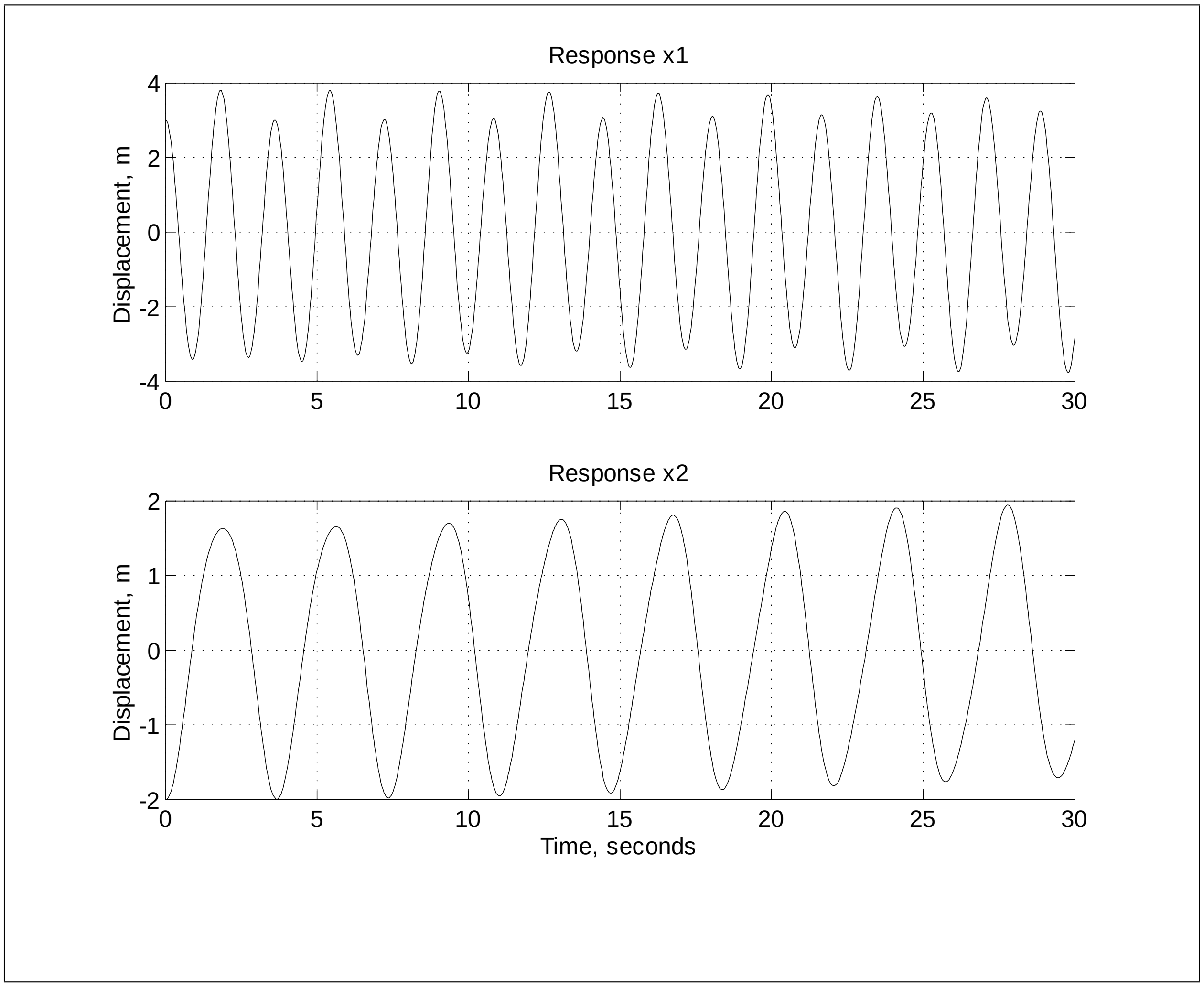

Figure 14.3: Response after changing initial displacement to [3 −2].

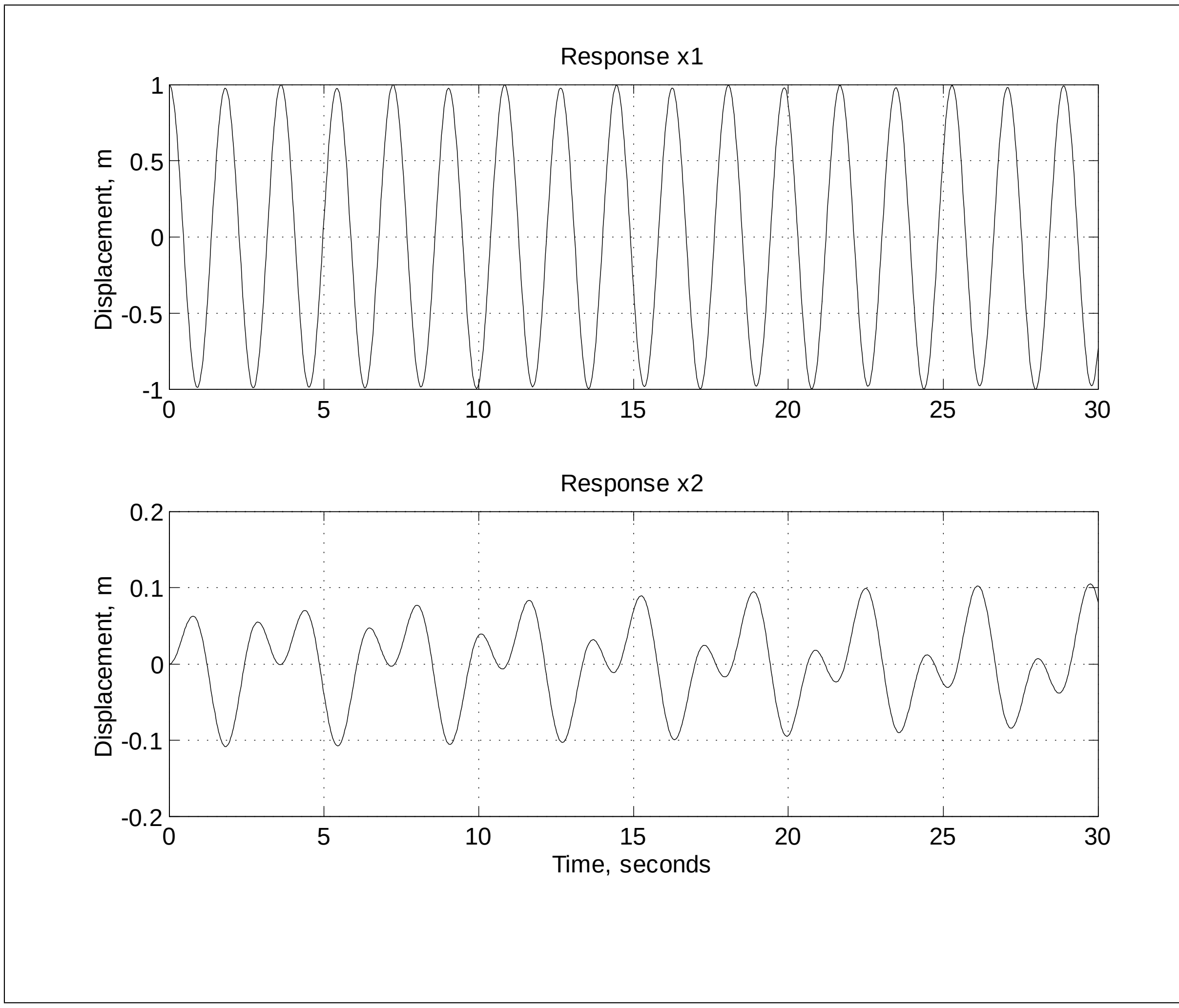

Figure 14.4: Response after changing initial displacement to [1 0].

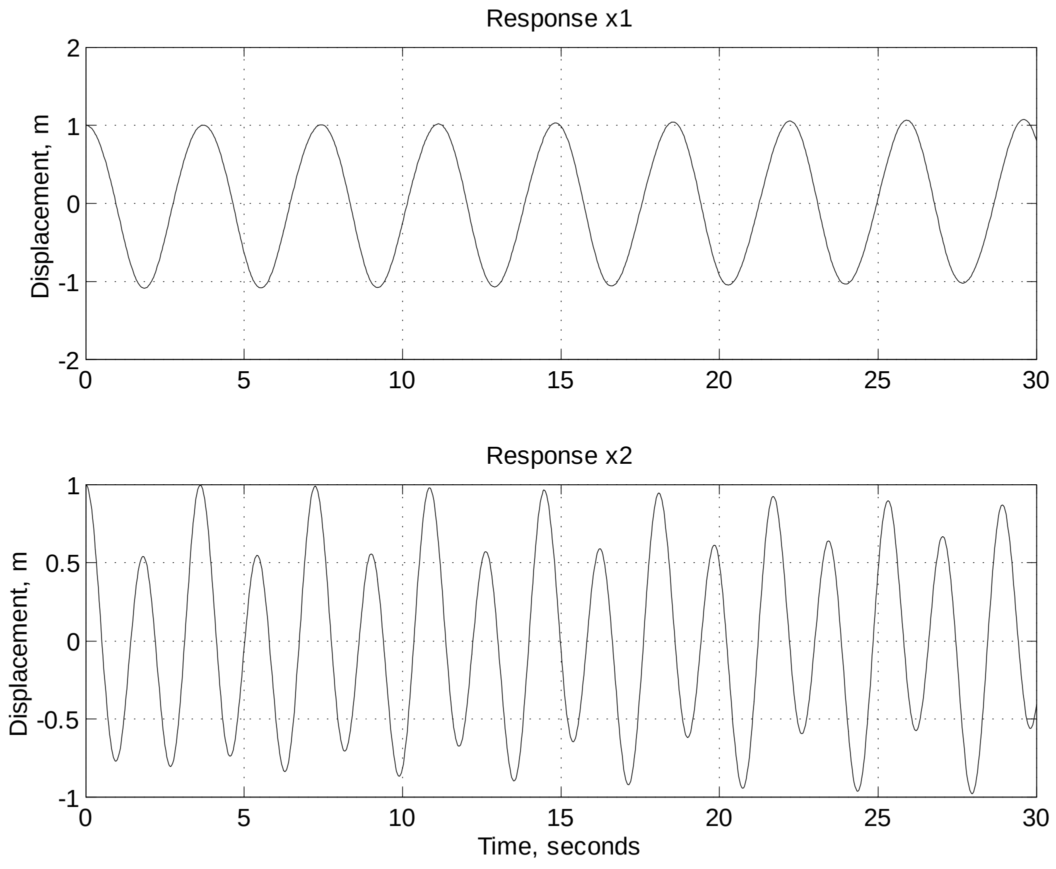

Figure 14.5: Response after interchanging system masses.

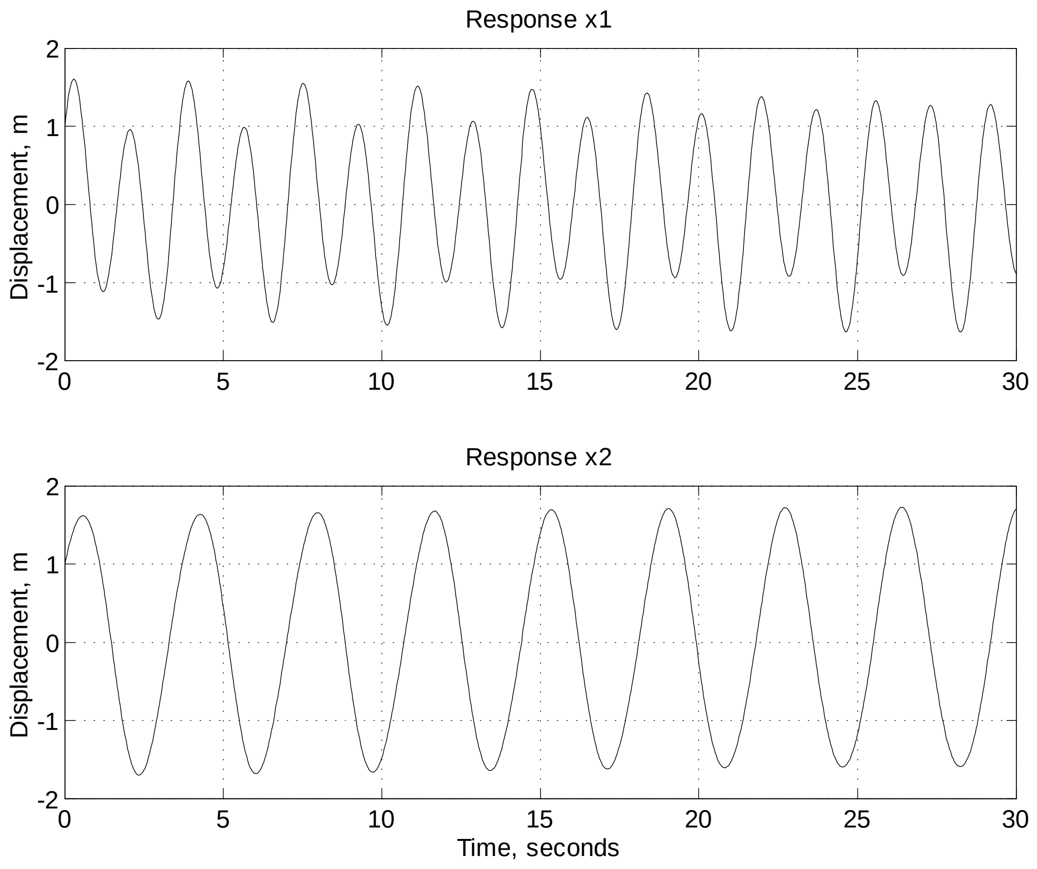

Figure 14.6: Response with initial velocity [4 2] and initial displacement [1 1].

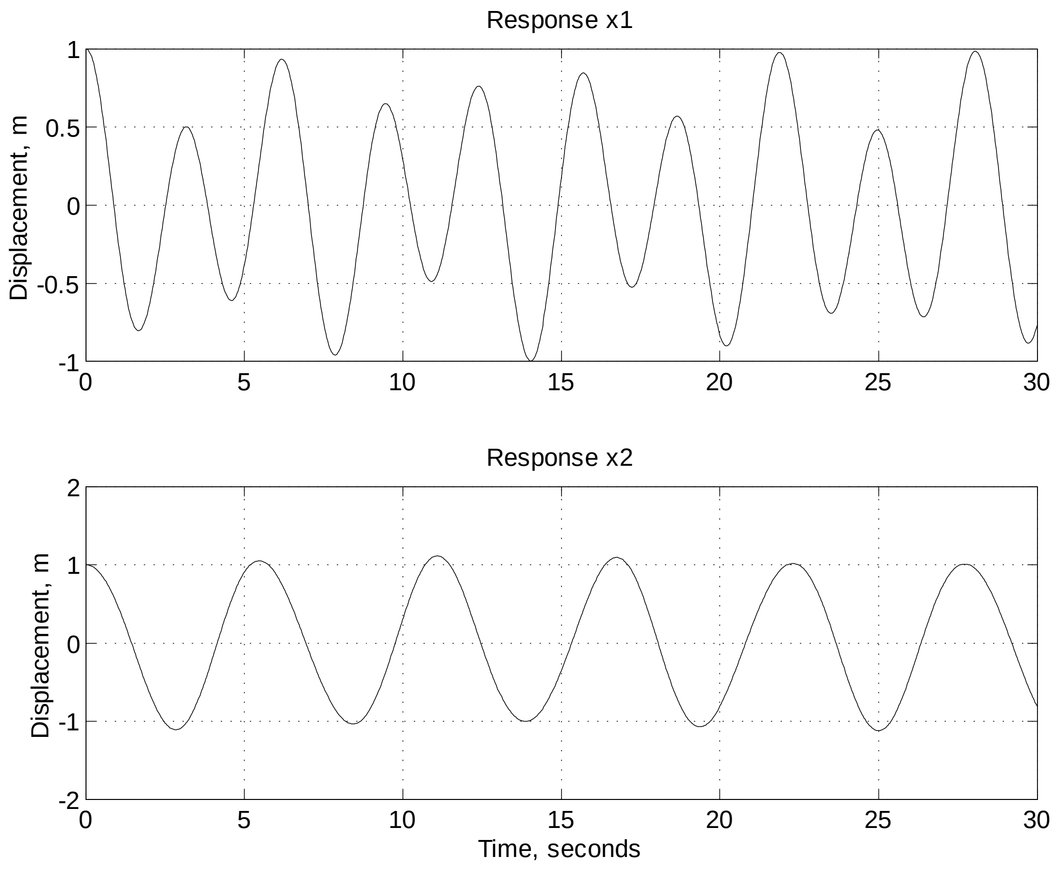

Figure 14.7: Response with increased system masses.

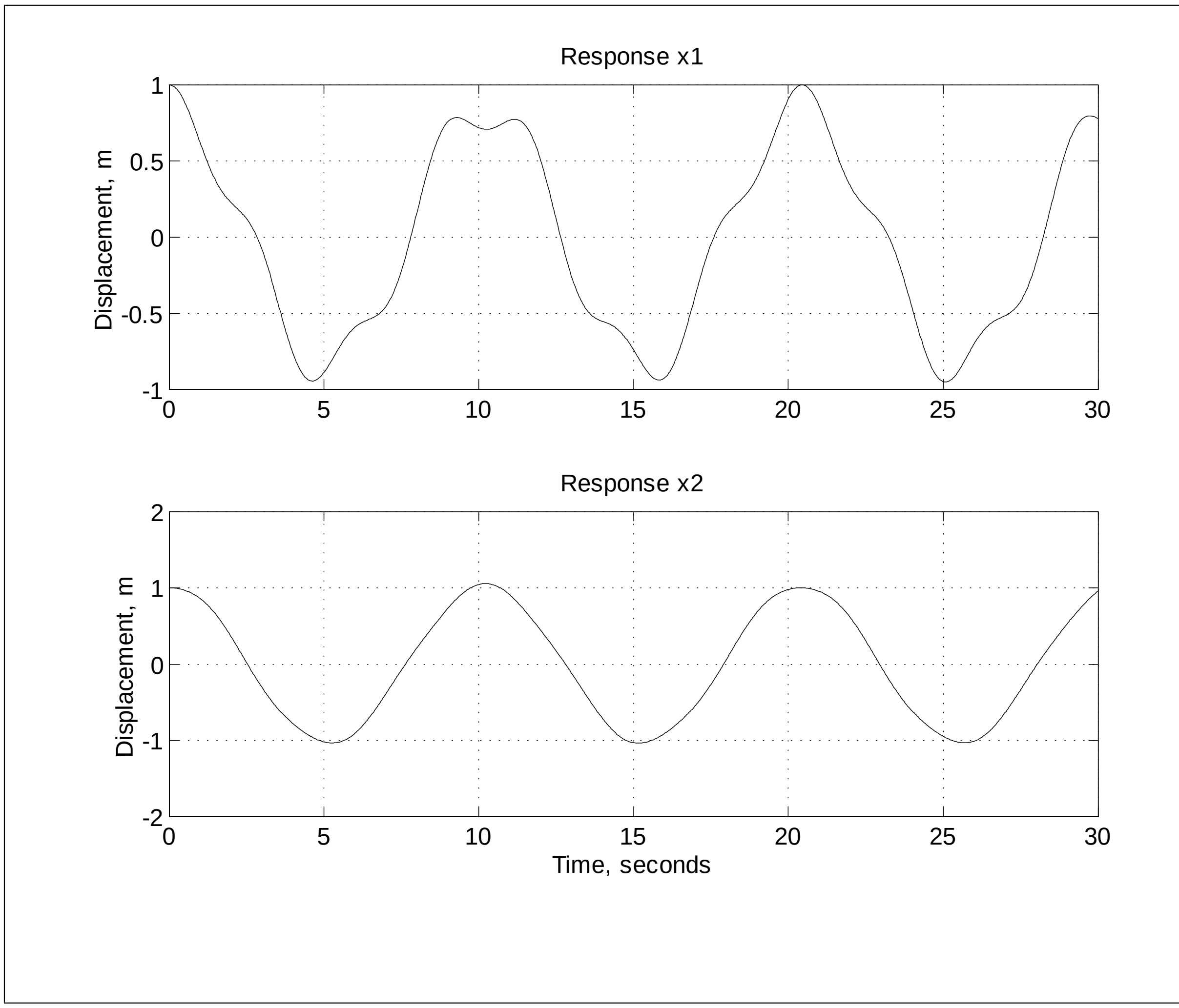

Figure 14.8: Response with reduced system stiffnesses.

Chapter 15

Damping of MDOF Systems

In the previous example, the free response of systems having more than one degree of freedom was discussed. Now, viscous damping is added to the system, as shown in Figure 15.1, and we will examine the changes caused by this addition. The equations of motion become:

$$m_1\ddot{x}_1 + (c_1 + c_2)\,\dot{x}_1 - c_2\dot{x}_2 + (k_2 + k_1)\,x_1 - k_2x_2 = 0 \tag{15.1}$$

$$m_2\ddot{x}_1 + c_2\left(\dot{x}_2 - \dot{x}_1\right) + k_2\,(x_2 - x_1) = 0. \tag{15.2}$$

In matrix form, we obtain:

$$M\ddot{\mathbf{x}} + C\dot{\mathbf{x}} + K\mathbf{x} = \mathbf{0},$$

where $M = \begin{bmatrix} m_1 & 0 \\ 0 & m_2 \end{bmatrix}$, $C = \begin{bmatrix} c_1 + c_2 & -c_2 \\ -c_2 & c_2 \end{bmatrix}$, $K = \begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix}$.

As we know, with damping present, there is no guarantee that the equations will decouple by the procedure used earlier. However, for proportional damping, i.e., $C = C_mM + C_kK$, it is possible to follow the previous decoupling procedure. This is because both K and M are diagonalized through the matrix of eigenvectors U. Thus, a scalar multiplied by M or K will also be diagonalized, and the sum of two diagonal matrices is also diagonal.

While proportional damping is something of a savior for modal analysis, it presents a headache when programming. The only way to assure decoupled equations in a program is to specify C_m and C_k instead of entering the matrix

C into the program, or to take the matrix C, go through the first two steps of modal analysis with it, and then check it for diagonality. Both of these methods have flaws; in the first case, the person using the program may not know C_m and C_k at all, and they may be difficult to calculate. For the second algorithm, the program may run through a significant amount of calculation to deduce that the given data is invalid. This wastes time and effort, especially if the data is invalid due to an error in entry of values. So, when programming, the most efficient way to handle damping is through the modal damping ratios (though the program given below retains some flexibility by allowing entry of the modal damping ratios or the factors C_m and C_k).

The modal damping ratios are simply an analogue for the damping ratios we saw earlier in our study of single degree of freedom systems. They are called modal damping ratios because they are applied independently to each modal (decoupled) equation; the modal damping ratios are automatically decoupled. Calling the modal damping ratios ζ_i, where i is the number of the mode being examined, the equations of motion in the modal coordinates become:

$$\ddot{q}_i + 2\zeta_i\omega_i\dot{q}_i + \omega_i^2 q_i = 0,\ i = 1, 2, \ldots, \tag{15.3}$$

where q is the modal coordinate variable, and ω_i represent the natural frequencies of each mode. This is the same as for the damped single degree of freedom oscillator, whose solution was:

$$q_i(t) = A_i e^{-\zeta_i\omega_i t}\sin(\omega_{di}t + \phi_i), \tag{15.4}$$

where ω_{di} is the damped natural frequency of that mode ($\omega_i\sqrt{1-\zeta_i^2}$), and

$$A_i = \left[\frac{\left(\dot{q}_{io} + \zeta_i\omega_{di}q_{io}\right)^2 + \left(q_{io}\omega_{di}\right)^2}{\omega_{di}^2}\right]^{1/2} \tag{15.5}$$

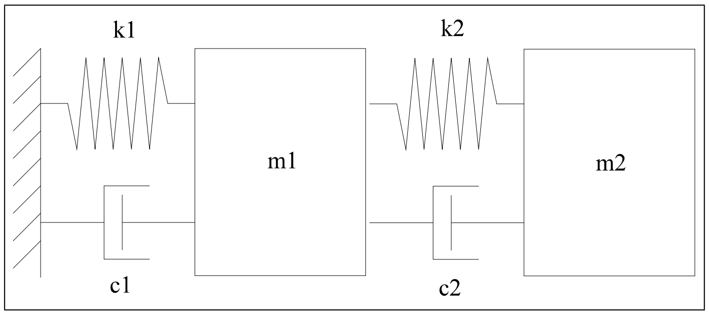

Figure 15.1: Two degree of freedom damped oscillator under consideration here.

$$\phi_i = \tan^{-1} \frac{q_{io}\omega_{di}}{\dot{q}_{io} + \zeta_i \omega_{di} q_{io}}, \tag{15.6}$$

noting that q_{io} and $\dot{q}_{io}$refer to the initial displacement and velocity in the modal coordinate, respectively. Having obtained the solution for $\mathbf{q}(t)$, we then use the last step of the modal analysis procedure to find $\mathbf{x}(t)$. Note that this method uses the entire modal analysis procedure as if damping did not exist, and then adds in the damping when solving for $q_i(t)$. The program at the end of this example, used to generate the figures that follow, demonstrates this approach.

Figures 15.2 through 15.4 use $m_1 = 9$, $m_2 = 1$, $k_1 = 24$, $k_2 = 3$, $\zeta_1 = 0.05$, and $\zeta_2 = 0.1$. Figure 15.2 uses an initial displacement vector of $x_0 = [1\ 0]$, and the results show the effect of the damping; the response gradually dies out. In Figure 15.3, an initial velocity of [3 2] is added, and the result is no increase in the transient period (as expected), but an increase in amplitude. Finally, Figure 15.4 removes the velocity and changes the initial displacement to [2 1], and the result is a greater amplitude, but the same length of transient period as Figure 15.3. In all three figures, note that both responses show the effects of multiple sinusoids, even though the first mass is much greater than the second. This is because the first stiffness is very large; this stiffness keeps the mass from gaining too much velocity, and so the smaller mass can effectively transfer its momentum to the larger.

Finally, Figure 15.5 changes the damping ratios to $\zeta_1 = 0.07$ and $\zeta_2 = 0.03$. The difference between this figure and Figure 15.2 is that here the responses have a greater amplitude and a larger transient period, due to the smaller damping ratios.

Program 14-1: moddamp.m

```
% This code solves n-degree of freedom damped
% vibration using the modal analysis techniques
% of Section 7.6.3.
%
clear;
M=input('Enter the mass matrix: ');
[n,o]=size(M);
if n~=o
error('M matrix must be square!');
end
K=input('Enter the stiffness matrix: ');
[n,o]=size(K);
if n~=o
error('K matrix must be square!');
end
% Giving the option to input damping via
% modal damping ratios or through the multiplying
% factors Cm and Ck.
%
fprintf('Press 1 to enter modal damping ratios, ')
qz=input('or anything else to enter Cm and Ck. ');
if qz==1
for iz=1:n
zeta(iz)=input(['Enter the damping for mode ',num2str(iz),': ']);
end
else
fprintf('Given that [c]=Cm[M]+Ck[K], ');
Cm=input('Enter the factor Cm: ');
Ck=input('Enter the factor Ck: ');
end
qu=0;
[u,l]=eig(K,M);
% Using "eig" in this way allows us to subtract M*w^2
% from K, instead of I*w^2 (where I is the n by n identity
% matrix).
% The output from "eig" gives unit-length eigenvectors.
% We need to scale them with respect to M.
%
for s=1:n
alfa=sqrt(u(:,s)'*M*u(:,s));
u(:,s)=u(:,s)/alfa;
end
x0=input('Enter the initial displacement column vector: ');
xd0=input('Enter the initial velocity column vector: ');
tf=input('Enter the final time: ');
```

```
t=0:0.1:tf; q=tf/0.1;
x=zeros(size(n,q));
for j=1:n
w(j)=sqrt(l(j,j));
% If modal damping ratios were entered, we already have
% a zeta vector. If not, we need to calculate from Cm
% and Ck.
%
if qz~=1
zeta(j)=0.5*(Cm/w(j)+Ck*w(j));
end
wd(j)=w(j)*sqrt(1-zeta(j)^2);
xt=u(:,j)*(u(:,j)'*M*x0*cos(w(j).*t)/sqrt(1-zeta(j)^2)+u(:,j)'*M*xd0/...
w(j)*sin(w(j).*t)/wd(j));
x=x+xt;
end
for i=1:n
x(i,:)=x(i,:).*exp(-zeta(i)*w(i).*t);
end
for r=1:n
subplot(n,1,r)
plot(t,x(r,:))
xlabel('Time, seconds');
ylabel(['Response x',num2str(r)]);
end
```

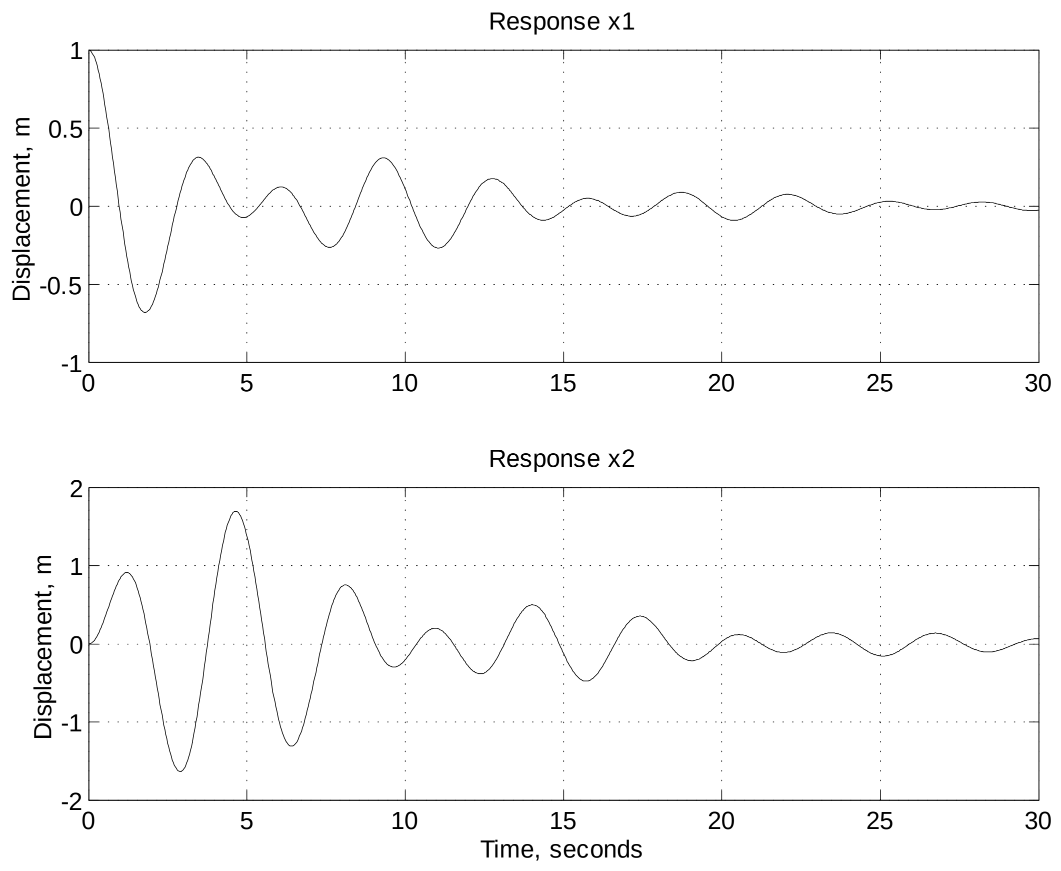

Figure 15.2: Damped response of a two degree of freedom system.

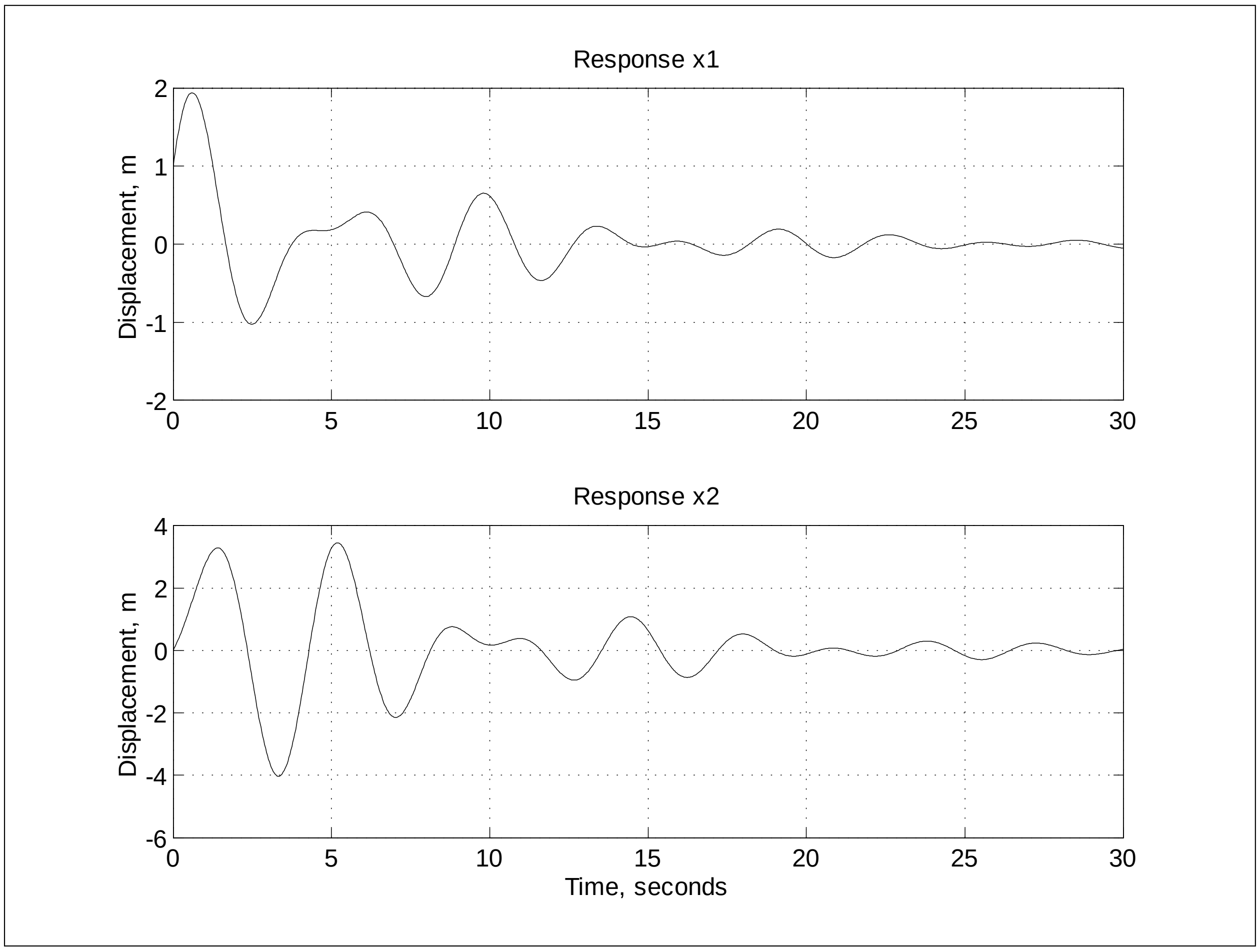

Figure 15.3: Response of a damped 2-DOF oscillator after adding an initial velocity.

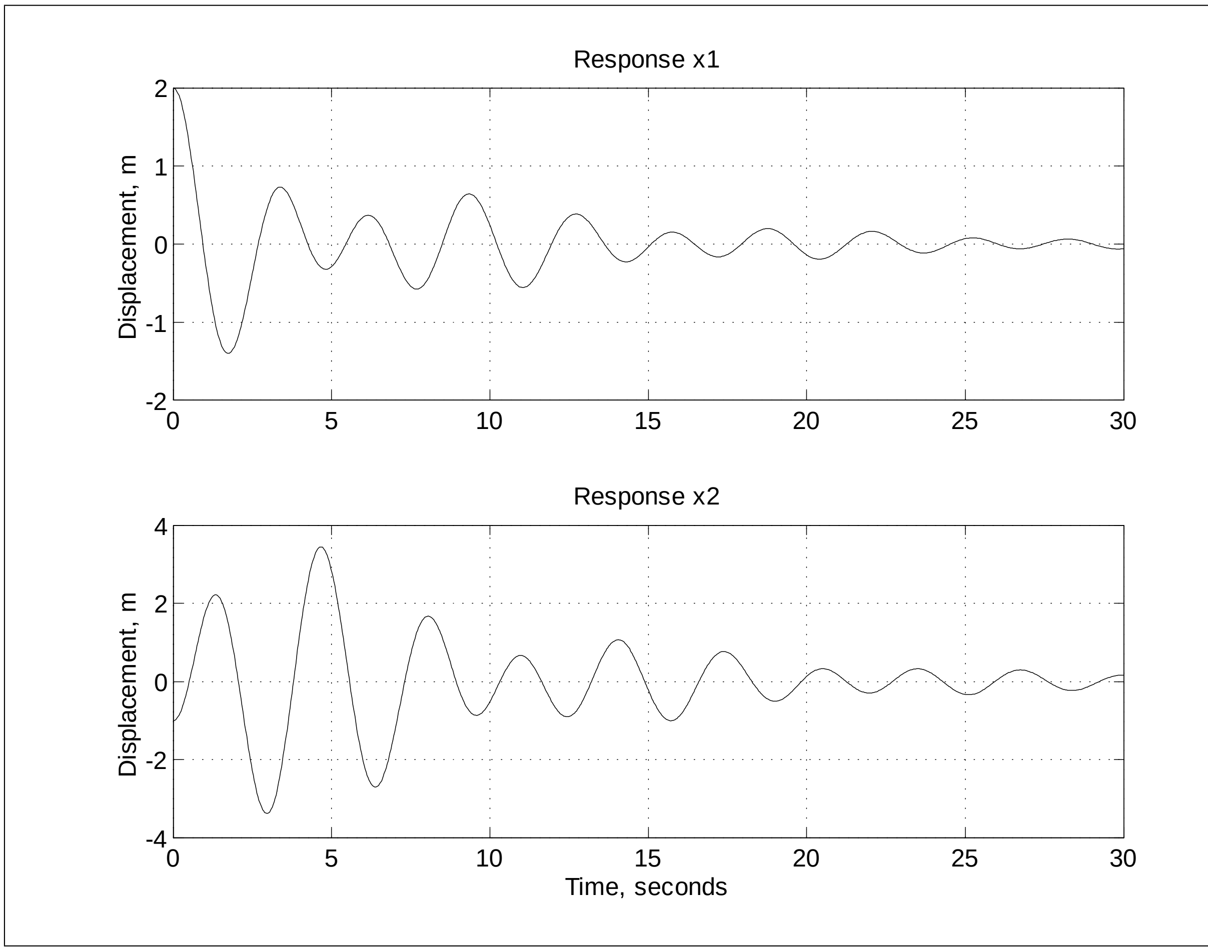

Figure 15.4: Response of the same system to a different initial displacement.

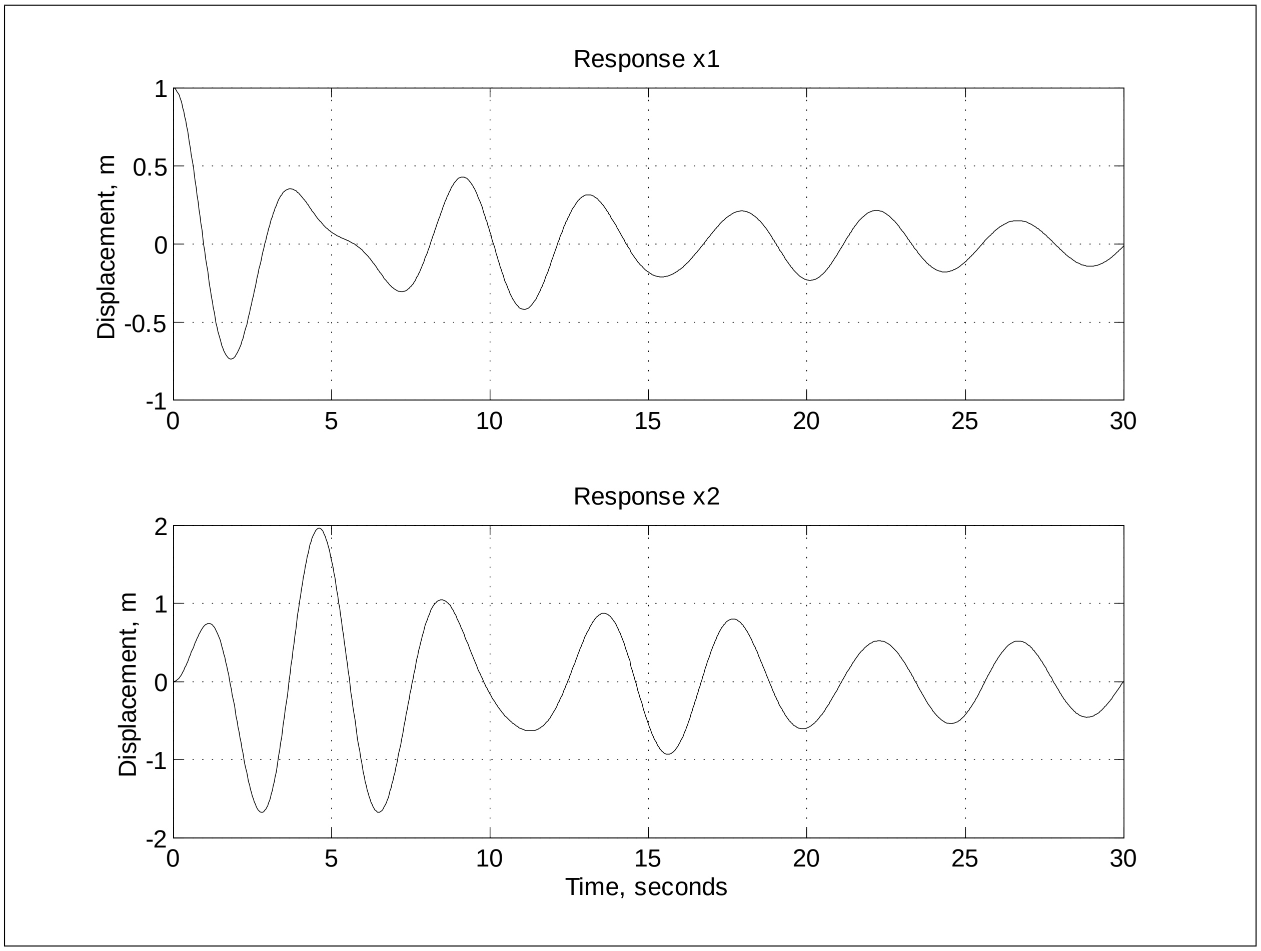

Figure 15.5: Response of the 2-DOF system with different modal damping ratios.

Chapter 16

A Universal Vibration Solver

The next logical step in this sequence of examples would now be to introduce forcing functions of different types (as seen earlier with the single degree of freedom cases), and create several different programs. While this method is effective, it is not efficient by any stretch of the imagination. The best way to proceed from here would be to create a program that solves all vibration cases, no matter what the forcing function or number of degrees of freedom.

The simplest way to go about this is to begin with a general program that does not include the external force. Program 1, *ndofold.m*, demonstrates this method. Some techniques used in this program demonstrate ways to make a program modular. We need a general program to be as modular as possible so that any changes that need to be made between runs of the program (like changing the masses or the stiffnesses) can be done without editing the code. The first major step in achieving this goal is the first line of the code. This line assigns a variable to the number of degrees of freedom. This way, in any loops or assignment statements made afterward, this variable can be referred to, and can represent any number of degrees of freedom.

The next several lines initialize the variables needed later. This is done in lieu of a *clear* statement, because the *clear* statement will remove some variable that will be needed later. Additionally, not using the *clear* command has the advantage of allowing the user to assign variable names to the desired mass, stiffness, and other matrices. This way, when the program prompts for these later, the user need only enter the variable name, not the entire matrix. If *clear* is used, the matrices cannot be created beforehand. The trade-off in not using *clear* is that MATLAB must be told how large the matrices are to be; if these sizes are not specified, MATLAB will think the variables are to be the size that they are already. If we are now solving a three degree of freedom problem, and we had just previously solved a 2-DOF system, MATLAB will think the mass matrix is to be 2 by 2. The fifth element entered would set off an error message. The command used fills each matrix with zeros; this is a harmless way to initialize variables. Note how the variable n is used throughout this portion; imagine the headache that would be involved if the program had not prompted for n earlier. Each time an n appears, a change would have to be made to allow a different number of degrees of freedom to be analyzed.

After the initialization section, the program prompts for each matrix that will be needed. Having the necessary information, the program then solves the system by modal analysis, and prints out the output. Note how each calculation is done in loops that run from 1 to n. This maintains the generality of the program, allowing the user to specify with the first input how large the matrices will be. Notice also that the plots are put up one at a time. This is done because there is no way to guarantee a subplot of any number of columns will be readable. Imagine a forty degree of freedom system, and you will understand why a subplot cannot be guaranteed. So, the *pause* command is implemented after each plot to allow the user to examine each plot to his satisfaction and to get a printout of the plot before the next one comes up (a printout can be obtained from the graphics window in MATLAB for Windows only). The program does not automatically print the graphs because they may be incorrect or uninteresting, and to print them in such a case would be a waste of time and paper. However, modifying the program to specify printing would take a few seconds (replace *pause* with *print* in the code). Also, a loop could be created at the beginning of the program to allow the user to specify whether output goes to the screen only, to a file, to the printer, or to a combination of the three. A sample of such a loop follows here:

```
%This loop allows the user to specify the nature of the output.
%Notice how the loop assumes that the user will enter the
%wrong response; this is to correct for any mistakes, and
%to thwart those who love to crash programs.
%Also, the program initializes the variable right before the loop
%to the value necessary to continue the loop.
outp=8;
while outp==8
    fprintf('Enter 1 to print to the screen only, 2 to print to a file, \n')
    outp=input('or 3 to print to the printer. ');
%Notice how the text was split across two lines by using
%the fprintf statement. This allows the text for input
%statements to be as long as needed.
%
    if outp~=1&outp~=2&outp~=3
%Translation: "If outp isn't equal to 1, 2, or 3."
%"~" means "not" for a MATLAB logical statement,
%and the ampersand (&) is "and", as usual.
%This is where we let the user know he's made a mistake.
        fprintf('Enter 1, 2, or 3!!!')
%Direct and to the point.
```

```
%Now, outp is reset to the value needed to continue the
%loop, since we need the loop to continue.
%
            outp=8;
      end
end
%One "end" for the "if", and one for the "for".
%A common mistake is to leave one out, and then
%MATLAB tells you that it came to the end of the
%file in the middle of a loop. (Not a pleasant feeling.)
```

So, there's the loop! It's not long (without the comments), and it's effective. It can be expanded as much as necessary to allow for any combination of printing options. One more thing: to get this to work, you need to put the following into the printout loop (where the *pause* is now):

```
if outp==2
%"==" in MATLAB is "equals" to you and me.
print -dps file1.ps
elseif outp==3
print
elseif outp==1
pause
end
```

Note that if you type "else if" instead of "elseif", MATLAB reads the "else if" as a new if statement, needing its own end. Another common mistake.

Now, it's just a matter of adding forcing functions to the mix. This task, however, is nearly impossible to do in a general way with the algorithm selected. So, we need to go back to the drawing board and find a simpler method. The first thing to realize is that the differences among the several possible vibration cases occur in solving the differential equations. Every problem in vibrations begins as a set of second-order differential equations, possibly coupled, possibly not. Fortunately, MATLAB has the *ode45* command, which solves differential equations numerically for a given time interval and initial conditions. Unfortunately, this command requires that the differential equations exist in a function M-file which defines the system of differential equations. Also, these differential equations must be first-order.

The problem of order is really not a difficulty at all; there is a method by which a second-order differential equation can be transformed into a system of 2 first-order equations. However, unless a general M-file can be created, the user would have to create his own function M-file each time, which would then be used for *ode45*. So then where would be the use of a program?

The answer lies in automating the process of creating an M-file. MATLAB has file transfer protocols, like any other programming language. We can write a program that tells MATLAB to create a specific M-file from a general procedure we write into the program, and then solve the resulting system of differential equations with *ode45.* Program 2, *ndof.m,* performs this task.

Again, the program begins by specifying the number of degrees of freedom for the run. The necessary matrices are initialized, and the inputs are obtained. After this, the program performs the first few steps of modal analysis, in order to find the natural frequencies. We can do this because the natural frequencies only depend on the mass and stiffness matrices; we know that these can be decoupled.

The next set of commands are initiated if a damping ratio matrix was entered. This loop changes the damping ratios into damping coefficients, by reversing the modal analysis procedure. This can be done because the damping ratio vector that the user enters is related to the diagonal of the decoupled damping coefficient matrix. Since this damping coefficient matrix is decoupled, it can be transformed from the modal coordinate system (the decoupled coordinate) back to the original coordinate system by applying the reverse of the modal analysis procedure.

Now that the program has the original values of the mass, stiffness, and damping coefficient matrices, the M-file can be created. The program first opens the file with the *fopen* command. The strings in parentheses by the command specify the filename, and the read/write privilege desired (‘`w+`’ means ’write and overwrite,’ which allows a user of this program to make several runs in succession without erasing the M-file previously created; the previous M-file is lost). The *fopen* command is set equal to a variable because the command specifies an arbitrary number by which MATLAB identifies the file. The only way to get this number is by setting *fopen* equal to a variable, so that this number will be stored in an accessible place.

The reason that we need this identification number is shown by the *fprintf* statements that follow the *fopen* command. Each statement begins with the file identification number, so that MATLAB knows to write the text to the file specified earlier, not to the screen. If the file identification number was not set equal to a variable, we would have no way to access it, and no way to write to the file.

Next, the functions defining the external forces on each degree of freedom are entered individually. This is necessary because a text string entered into a MATLAB variable is stored one character per matrix element. So, if all the external forces were entered at once, there would be no way of telling how long the force variable would be, or how many elements the matrix would have. The method used in the program avoids these difficulties.

Notice the use of the *num2str* command within the *fprintf* statements. This command allows the value of a variable to be used in a text statement as text; it is possible to use the value of a variable in an *fprintf* statement, but not as text. Note that to use the *num2str* command, the text to be printed must be within brackets as well as parentheses; this is because, if the brackets are not included, MATLAB thinks the statement ends when the quotes are closed, and will produce an error instead of including the desired value of the variable.

Finally, the program uses an *fclose* statement to tell MATLAB that nothing else is to be entered into the file. This is set equal to a variable, because MATLAB indicates a successful write transfer through the *fclose* statement; a user can tell if the file was written successfully by seeing the value of this variable.

Now, the program takes initial values necessary for *ode45*, and solves. The plots come printed individually; a print loop similar to the one described above can be inserted into this code to allow printing to a printer or to files.

Figures 16.1 and 16.2 show the results of using *ndof.m*. Each figure represents a degree of freedom of the system, whose parameters are as follows: $m_1 = 9$, $m_2 = 1$, $c_1 = 2.7$, $c_2 = 0.3$, $k_1 = 27$, $k_2 = 3$, $F_1 = 0$, $F_2 = 3\cos 2t$. Notice how each degree of freedom shows the transient response over the first two seconds, and then settles into a steady-state response based on the single forcing function. The natural frequencies calculated by the program are $\omega_1 = 1.41$ and $\omega_2 = 2.00$, which match those calculated by hand. Also, the responses of each mode are found through manual calculation to be:

$$x_1(t) = 0.2451\cos(2t + 0.1974) - 0.6249\sin(2t) \tag{16.1}$$

$$x_2(t) = 0.7354\cos(2t + 0.1974) + 1.8749\sin(2t). \tag{16.2}$$

Looking at the maximum magnitudes of the responses shown in Figures 16.1 and 16.2, we see that they are approximately equal to the sum of the coefficients of the sine and cosine terms in the expression above, as we would expect. So, from this example, we see the power of the program and, more importantly, verify that the program works.

Program 16-1: ndofold.m

```
%This program calculates the damped response of
%multi-degree of freedom systems without external forcing.
%The user must enter the modal damping ratios; no other
%method of describing damping is supported.
n=input('How many degrees of freedom are present? ');
m=zeros(n,n);
k=zeros(n,n);
s=zeros(n,n);
p=zeros(n,n);
l=zeros(n,n);
ac=zeros(1,n);
phi=zeros(1,n);
m=input('Enter the mass matrix. ');
ko=input('Enter the stiffness matrix. ');
zeta=input('Enter the damping ratio (zeta) matrix. ');
x0=input('Enter the initial displacements. ');
xp0=input('Enter the initial velocities. ');
ti=input('Enter the initial time. ');
tf=input('Enter the final time. ');
t=ti:(tf-ti)/1000;tf;
[b1,b2]=size(t);
x=zeros(n,b2);
r=zeros(n,b2);
a=m^(-1/2);
kt=a*ko*a;
[p,l]=eig(kt);
s=a*p;
si=inv(s);
r0=zeros(n,1);
rp0=zeros(n,1);
r0=si*x0';
rp0=si*xp0';
w=sqrt(l);
wd=zeros(size(w));
for k=1:n
    wd(k,k)=w(k,k)*sqrt(1-zeta(k)^2);
end
%Note that the same loop control variable can be used for
%every loop, so long as the loops are not nested within
%each other.
for k=1:n
    ac(k)=sqrt(wd(k,k)^2*r0(k)^2+(rp0(k)+zeta(k)*w(k,k)*r0(k))^2)/wd(k,k);
    phi(k)=atan2(wd(k,k)*r0(k),(rp0(k)+zeta(k)*w(k,k)*r0(k)));
    r(k,:)=ac(k)*exp(-zeta(k)*w(k,k)*t).*sin(wd(k,k)*t+phi(k));
end
```

```
x=s*r;
clg
for k=1:n
      plot(t,x(k,:))
      o=num2str(k);
      title(['Response x',o])
      xlabel('Time, seconds')
      ylabel('Displacement')
      grid
      figure(gcf)
      pause
end
for k=1:n
      fprintf('Natural frequency w%g=%g \n',k,w(k,k))
end
```

Program 16-2: ndof.m

```
%This program, as stated above, writes its own
%M-file to use in ode45. This algorithm allows
%any external force combination to be examined,
%and also allows the external force to be
%a user-defined function.
n=input('How many degrees of freedom are present? ');
x=zeros(1,2*n);
ct=zeros(n,n);
m=input('Enter the mass matrix. ');
aw=2;
while aw==2
      fprintf('Press 0 to enter a damping coefficient matrix, or')
      aw=input('press 1 to enter a damping ratio (zeta) matrix. ');
      if aw==0
            c=input('Enter the damping coefficient matrix. ');
      elseif aw==1
            zeta=input('Enter the damping ratio matrix. ');
      else
            fprintf('Please enter 0 or 1.')
            aw=2;
      end
end
k=input('Enter the stiffness matrix. ');
a=m^(-0.5);
kt=a*k*a;
[p,q]=eig(kt);
omega=sqrt(q);
if aw==1
      for zt=1:n
            ct(zt,zt)=zeta(zt)*2*omega(zt,zt);
```

```
        end
        af=a^(-1);
        c=af*ct*af;
end
fid=fopen('pdx.m', 'w+');
fprintf(fid,'function pdx=pdx(t,x)\n');
for i=1:n
        fprintf(fid,['pdx(',num2str(i), ')=(-(']);
        for j=1:n
                if k(i,j)==0
                else
                        fprintf(fid,['(',num2str(k(i,j)),')*x(',num2str(j+n),')']);
                end
                if c(i,j)==0
                else
                        fprintf(fid,['+(',num2str(c(i,j)),')*x(',num2str(j),')']);
                end
        end
        f=input(['Enter F',num2str(i), '(t) in quotes, like '3*cos(2*t)'. ']);
        fprintf(fid,[')+(',num2str(f),'))/(',num2str(m(i,i)),');\n']);
end
for i2=1:n
        fprintf(fid,['pdx(',num2str(i2+n),')=x(',num2str(i2),');\n']);
end
status=fclose(fid);
v0=input('Enter the initial velocities as a row vector. ');
x0=input('Enter the initial displacements as a row vector. ');
ti=input('What is your initial time? ');
tf=input('What is your final time? ');
init=[v0,x0];
[t,ny]=ode45('pdx',ti,tf,init);
for kl=1:n
        plot(t,ny(:,kl+n),'b');
        title(['Response x',num2str(kl)]);
        xlabel('Time, seconds');
        ylabel('Displacement, m');
        grid
        figure(gcf)
        pause
end
for l=1:n
        fprintf('Natural frequency w%g=%g. \n',l,omega(l,l))
end
```

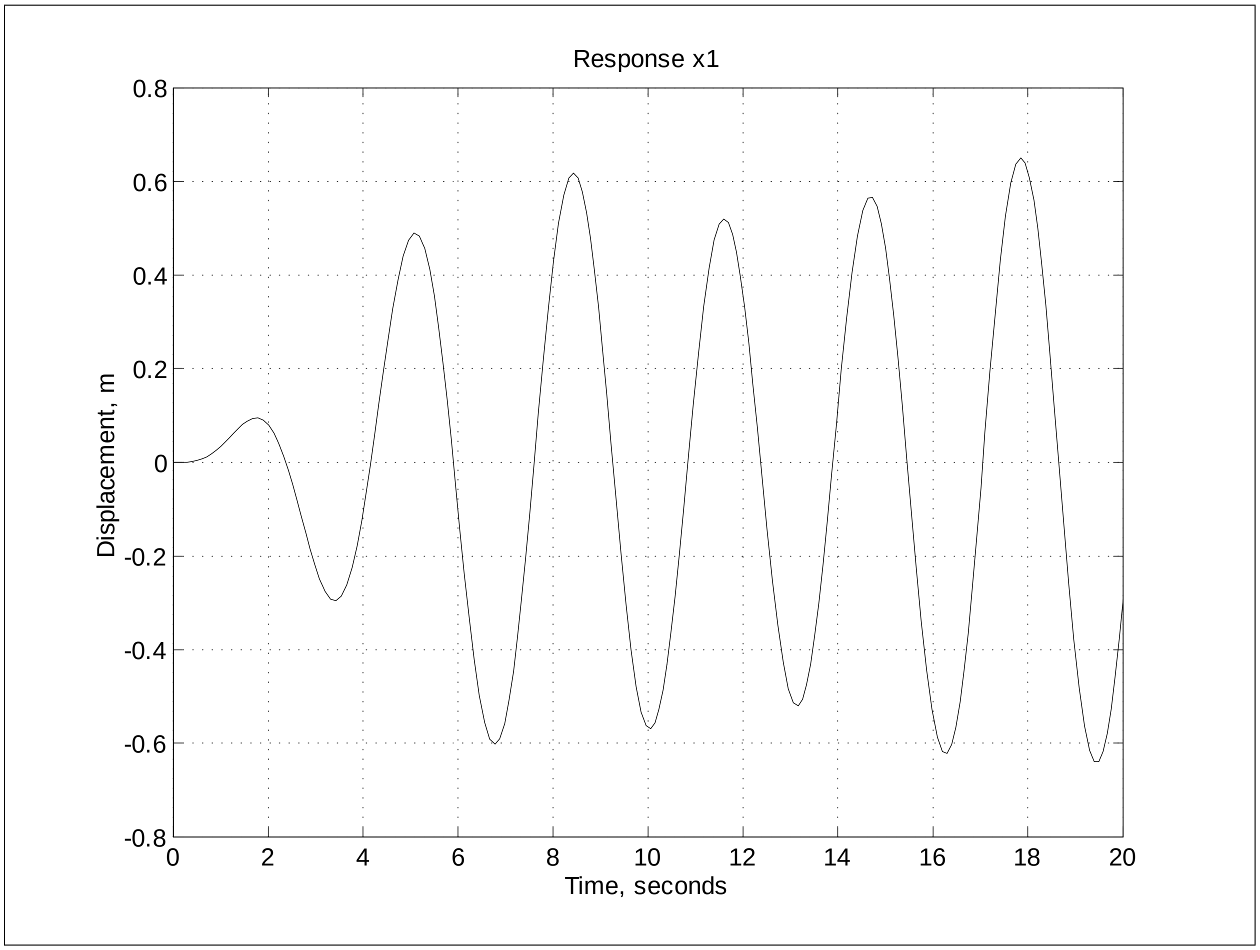

Figure 16.1: Response of the first mode.

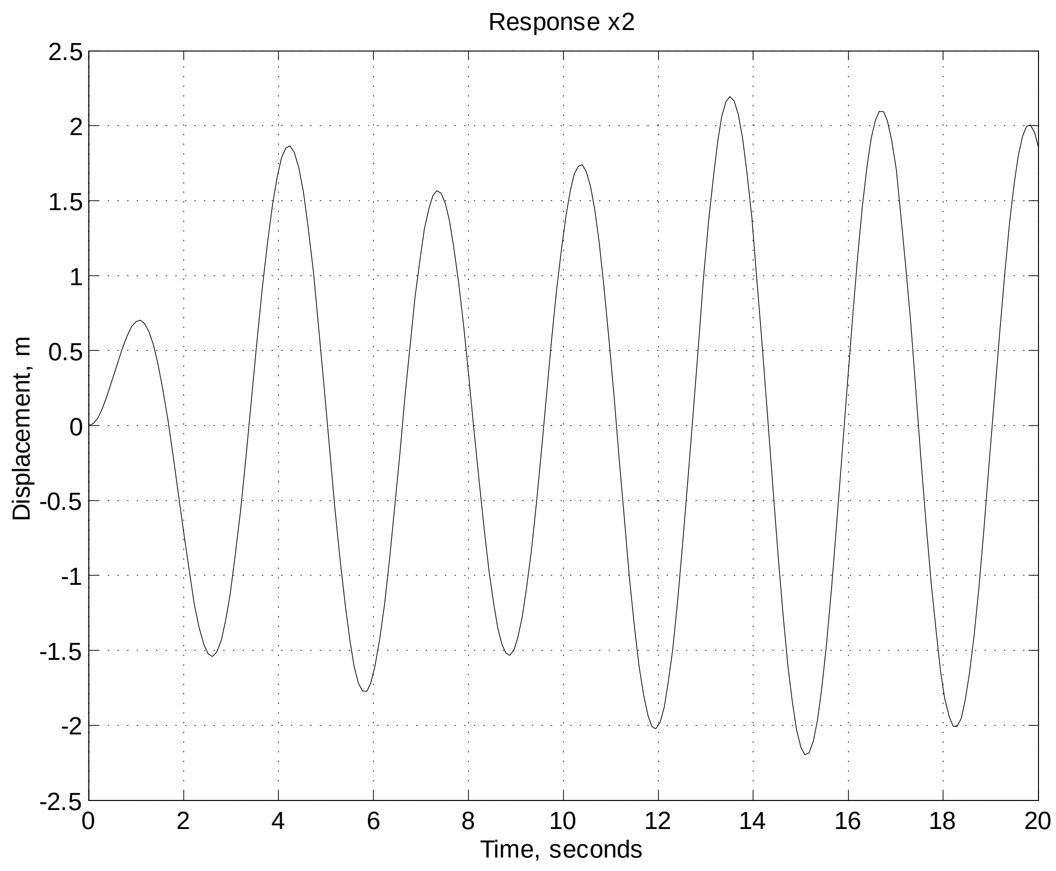

Figure 16.2: Response of the second mode.

Chapter 17

Modeling a van der Pol Oscillator

The next several examples will examine how we can use Matlab to solve for more real-world vibratory systems. The first such system we will look at is the oscillatory motion of a structure surrounded by a fluid. Examples of such structures include the support pylons of offshore oil platforms and antennae attached to the exterior surfaces of aircraft. These structures exhibit vibratory motion due to the creation of vortices in the fluid by viscous interaction between the structure and the particles comprising the fluid. Several investigators have applied the van der Pol equation,

$$\ddot{x} + e\left(x^2 - 1\right)\dot{x} + x = 0, \quad e > 0 \tag{17.1}$$

as a model for the motion of such a structure. Note that this equation, given the positive parameter e, exhibits the usual form of damping when $|x| < 1$. However, when $|x| > 1$, the term multiplying $\dot{x}$ will become negative. If we were to solve for $\ddot{x}$, then, the damping term would *add* energy to the system, instead of removing it. This *negative damping* approximates some of the phenomena observed in such fluid-structure interactions, and so is an attractive (and necessary) feature of the model. (The text gives a brief discussion of this problem as well, in Section 2.4.2.)

We will demonstrate how we can use Matlab to help us visualize the solution to such a problem. First, we must solve the nonlinear ordinary differential equation given above. Unfortunately, we will not be able to arrive at an analytic solution, as we did with several of the earlier problems we examined. What we will be forced to do, then, is to integrate the equation analytically over a particular time interval.

At first glance, it appears that we should be able to use a program like the general solver in the previous example. This impression is correct, but to use such a program would be wasteful. Why would we use a program that was designed for a general case, having many varied inputs, when the only parameters we will have to use here are the parameter e, the initial conditions, and the time step?

The trouble then becomes, how are we going to allow the parameter e to be dynamically assigned to the function in question? Our answer will come from Matlab's `global` variable capability. We will also define our van der Pol oscillator solver as a function, so we can demonstrate another useful technique to generalize Matlab functions. We will make e_g into a global variable, so that it can be passed freely between the main function and the function evaluating the van der Pol equation. (We will explain why we do not simply make e into a global variable in a moment.) This is illustrated in the code below; Program 1 is the main function, and Program 2 is the function used by `ode45` to solve the equation. We also introduce the `nargin` statement with this program. This command reads the input to the function, evaluating how many input arguments have been passed into it (you can think of it as Number of ARGuments INput). We define the function to take three inputs, but by using `nargin`, we allow as few as zero inputs to be passed to the function, providing default values for each input parameter. This allows our function to be quite flexible; if we're only interested in looking at the effects of changing the parameter e, we need not remember to pass in the same initial conditions and time span. We can use the default values provided by the function in each case.

To assign these default values, we test the value of `nargin` three times. If this value is zero, the three `if` statements provide all of the default input data. If it is one, the function uses the default values for the initial state and the time span. If the number of arguments is two, the function uses only the default time span. (Experienced C programmers may be wondering why we chose three `if` statements instead of a `switch` statement. The reader is invited to recast the `nargin` processing portion of this program to use the `switch` operator, and see how the program changes.)

Our trick to make this function work is to declare the parameter e_g as `global`. Note that there are two `global` statements in the programs below; one at the start of the main function, and one at the beginning of the function used by `ode45`. Also note that there are no semicolons at the end of these statements; there is no output to suppress, so semicolons are unnecessary. We are then able to use e_g in the subfunction as if it was assigned there, and not in the parent function, after initializing e_g to the value of e. This construction is needed because the function to be evaluated by `ode45` must be a function of the state and the time; adding in an extraneous third parameter will lead to errors. We could define every variable we use as `global`, but that would uneccessarily clutter the workspace with variables, and is considered poor technique. Global variables should be used with caution, as they allow parameters inside of a function to be changed transparently (for example, we could write Program 2 to adjust the value of e, effectively defeating the input value). The benefit of creating this van der Pol solver as a function is that the only variables created in the workspace are the outputs t and x; not even the global variable e_g remains in the workspace after function termination.

Now, why did we go to the trouble of introducing a spare parameter e_g to be our global variable? The answer comes from how Matlab handles its global variables. If you were to change e_g to e in the given code, you would find no trouble for the default case (no input arguments). However, if you called the function with a specific value of e, Matlab would print a warning statement saying that the value of your local variable (e) "may have been changed to match the global." This is because Matlab sees the variable e that is input to the function as a variable local to the function, and then sees the same variable declared as global later. Since the tail end of the warning includes the ominous statement "Future versions of Matlab will require that you declare a variable to be global before you use that variable," this author thinks it prudent to write his code in a way that will be forward-compatible, where possible. That means that our global variables will not be used before they are declared. In general, we will set apart our global variables with a trailing "g," like in the program below. The curious reader is encouraged to rewrite the code to use e as a global parameter, generating the Matlab warning.

The two figures below show some results for the default case (as defined in Program 1). Figure 17.1 shows a simple comparison of displacement versus time; we see oscillatory behavior. The other two figures that follow are *phase diagrams*[1], which are plots with the velocity on the y-axis and displacement on the x-axis. These diagrams are often used in studies of nonlinear and chaotic systems, since they can clearly show the effects of initial conditions on the response.

The key feature of Figure 17.2 is the closed loop. Note that the initial location is denoted by a circle, and the final state by a triangle. The closed loop means that the van der Pol system eventually settled down into oscillatory behavior. We can prove to ourselves that a closed loop in phase space corresponds to an oscillating response by considering the function $x = \sin(t)$. If we take this (obviously oscillating) function as our displacement, then the velocity is described by $y = \cos(t)$. If we were to plot this result in phase space, we would arrive at a circle; these functions $x(t), y(t)$ are parametric equations for a circle. The van der Pol oscillator's loop is not precisely circular, so it is not periodic in the same regular way as the sine or cosine function. However, it will repeat the same sets of positions and velocities.

Figure 17.3 is the same default system with different initial conditions; we used the command $[t, x] = solvevdp(0.5, [-1, 5])$. Again, this initial point ($[-1, 5]$) is denoted by a circle, and the final point by a triangle. Note that, in spite of the different initial conditions, the motion settles into the same limit cycle. Thus, the limit cycle is determined by the parameter e, and not by the initial conditions. This limit cycle behavior is similar to a phenomenon seen in the vibration of structures in a moving fluid; thus, several investigators have used the van der Pol equation to describe these systems. The reader is encouraged to try different initial conditions to see that the limit cycle is indeed a universal of the system.[2]

[1] Also called *phase portraits* by some authors.

[2] However, changing the parameter e will certainly change the form of the limit cycle in

```
function [t,x]=solvevdp(e,x0,tspan)
% Program 1: solvevdp.m
% This function takes in a value for e in the
% van der Pol oscillator expression,
% x”+e(x^2-1)x’+x=0, where primes denote
% differentiation in time. The function
% will automatically assume x(0)=1, x’(0)=0
% and a thirty-second time interval (tspan=[0 30])
% if these values are not provided. Also, a
% value of e=0.5 is assumed if none is given.
global e_g
if nargin<1 e=0.5; end % An “if” statement on a single line.
if nargin<2 x0=[1 0]; end
if nargin<3 tspan=[0 30]; end
e_g=e;
%
% Off to the ode solver.
%
[t,x]=ode45(’dxvdp’,tspan,x0);
function dx=dxvdp(t,x);

global e_g
% Program 2: dxvdp.m
% This is the function called on by ode45 to provide the
% derivative of our state vector x. We need to set up the
% second-order differential equation as a system of two
% first-order equations. Our state vector x consists of
% the variable x and its time derivative.
%
dx(1,1)=x(2);
dx(2,1)=-(e_g*(x(1)^2-1)*x(2)+x(1));
```

phase space.

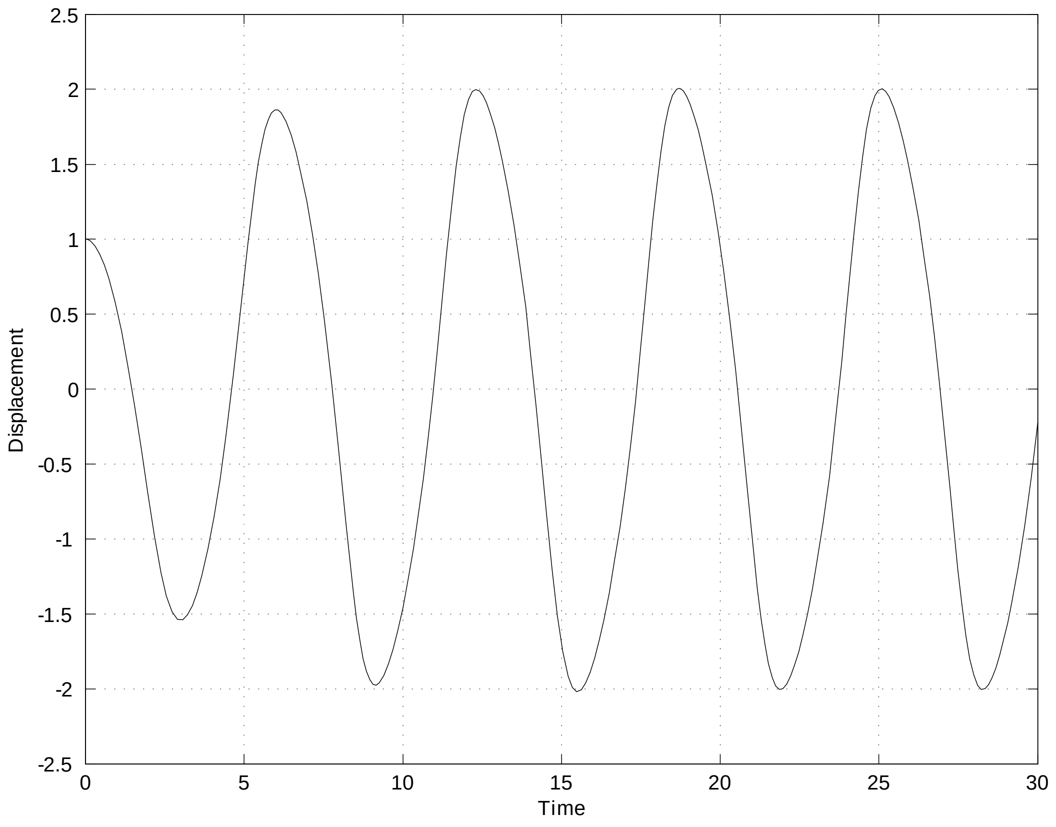

Figure 17.1: Displacement vs. time curve for the van der Pol oscillator under the default conditions of Program 1.

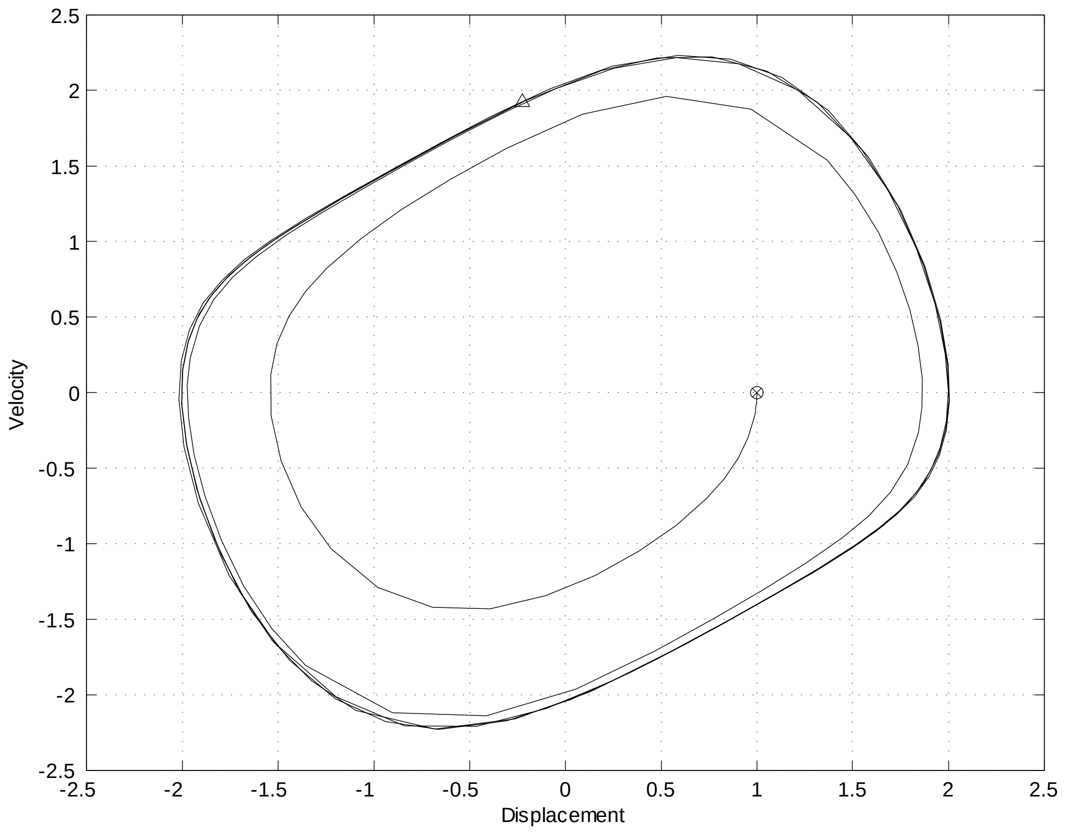

Figure 17.2: Phase diagram for the van der Pol equation. Note the closed limit cycle.

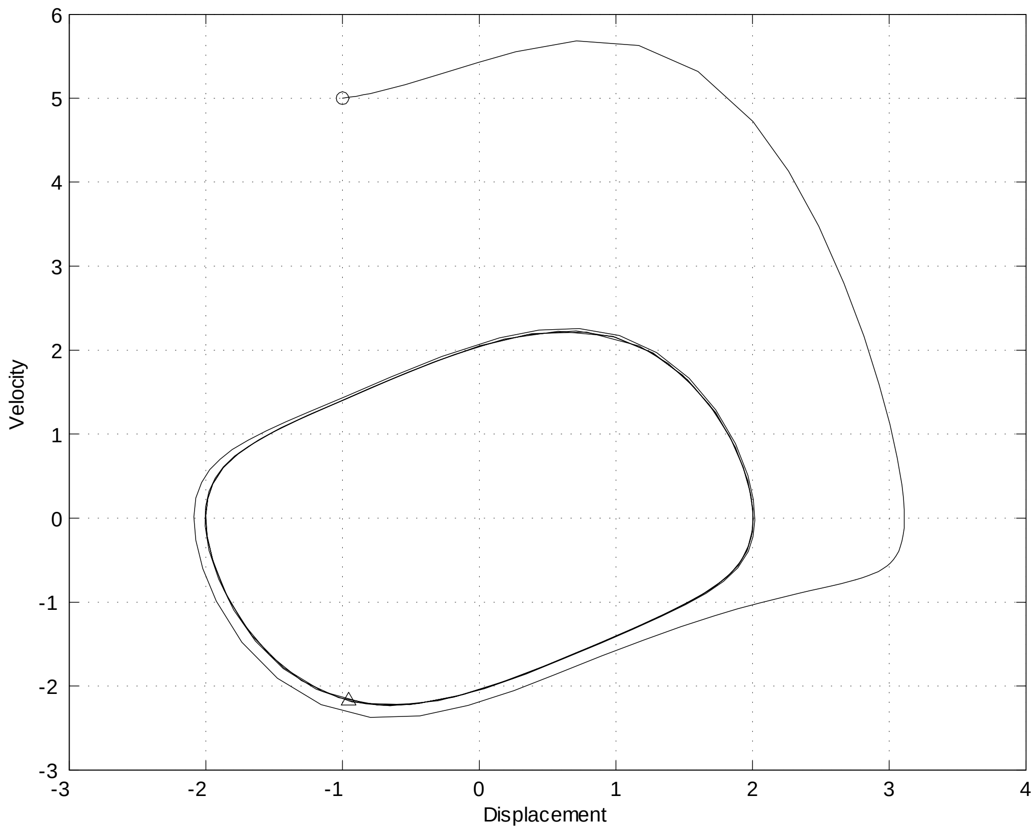

Figure 17.3: Phase portrait of the same van der Pol oscillator with different initial conditions.

Chapter 18

Random Vibration and Matlab

The focus in the text is probabilistic modeling of forces in vibrating systems. Our aim here is to show how Matlab can be used as an aid in modeling random processes, and thus in visualizing the response of systems subjected to random forcing.[1] The logical starting point for this effort is to show how we can create random forcing histories to apply to a deterministic model. Unlike the problems posed in the text, we will not be looking for the response statistics at this time; instead, we will arrive at sample response realizations, which we will later use to arrive at the response statistics numerically, contrasting the analytic approach taken in the chapter.

To illustrate some of the challenges associated with a random forcing input, let us look at a sinusoidal forcing input of the form $F(t) = A\cos(\omega t)$. We first need to determine how we will introduce randomness to this system. That is, will we apply a random force amplitude, a random frequency, or both? If we are modeling a given physical system, we will find that this question often answers itself. For example, we may have derived the power spectral density of a typical input to our system, and found that the vast majority of the energy input to the system comes at one particular frequency. For this case, we could then (as a first approximation) treat the forcing frequency as deterministic, and use a random distribution for the amplitude.[2]

Having determined the parameter we will consider to be random, we need to arrive at a probability density function for our this parameter. That is, will we use a uniform distribution, a Gaussian distribution, or some other description? Before we decide this, let us have a look at the distributions available inside Matlab.

[1]Following the example of the textbook, we will limit ourselves mainly to random forcing of deterministic systems.

[2]In general, we would also have a random phase angle associated with this forcing. We omit the phase angle here for simplicity of analysis.

The uniform and Gaussian distributions are built into Matlab through the `rand` and `randn` commands, respectively.[3] Typing `a=rand(n)` at the Matlab prompt will return an $n \times n$ array of randomly-generated numbers in a, using a uniform distribution on the interval [0.0,1.0]. The `randn` command returns a random number from a Gaussian distribution having a mean of zero and standard deviation of one. Note that Matlab's random numbers are actually "pseudo-random"; that is, they depend on an initial state vector. For some applications, we will want to compare the results of several different systems to a random loading. For instances like this, we can use the state of the random number generator to subject each comparison case to the same realization of the random loading process. If we want to ensure that we get a different realization each time, we can use the suggestion available in the Matlab help for `rand`, and initialize the state to `sum(100*clock)`. This will reset the random number generator based on the current time.[4] Both methods are useful for different cases. Any other probability distributions would have to be derived from the two described above or created from scratch by the user.

Following the lead of Section 4.8, let us look at the system described by

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = F(t) \tag{18.1}$$

where we will take $F(t)$ to be the sinusoid above having random amplitude $A(t)$. It will be convenient for us to apply the condition of ergodicity to this system. Thus, we will solve for the *steady-state response* only, neglecting the transient response. (This also eliminates the need to specify initial conditions.) We can solve for the steady-state response in two ways using Matlab. We can use the results of the analysis we did earlier, giving an analytic solution, or we can use the numerical integration routines. We'll do the latter in this case, to show how we can transport the amplitude values back and forth where they are needed, and to show how we can handle the discrete datapoint set.

Again, we use the `nargin` function to allow any number of input parameters from zero to the maximum of five. Recall from the last example that we used less than statements instead of a `switch` expression to define parameters. This is because expression cases of a `switch` in Matlab do not require a `break` statement to end each expression set, unlike the C analogue. Thus, we cannot allow cases for lower numbers of input arguments to "fall through" to higher numbers, like we could in C. We also set the length of the amplitude vector A to be the same as the length of the *tspan* vector, or 1000 elements if the input *tspan* is shorter. We do this to make interpolation of A based on values of t simpler. When Matlab integrates our differential equation, the values of t that it will test are not necessarily the same as the values in *tspan*. Thus, we will interpolate linearly between neighboring values of A corresponding to the two elements

[3]There also exist three other commands, two of which produce sparse matrices, while the third gives a vector with a random permutation of the first n integers. These are not useful to us here.

[4]Specifically, the `clock` command returns a six-element vector, containing the current year, month, day, hour, minute, and second.

of *tspan* that surround the current time value t.[5] In order to ensure that the amplitude is always greater than zero, we multiply each element of A by its sign. This leads to a distribution that is not rigorously Gaussian. We could improve our odds of not getting negative values for A by changing the standard deviation such that 3σ in each direction is positive. The coding for this is left to the reader. The `error` expression in the default (`otherwise`) case of our `switch` expression will print the message in quotes to the screen, informing the user of the function that the allowed values of the flag are zero and one.

We are confronted with one more problem if we wish to use the differential equation solver. That is, how do we remove the transient response? It is not as simple as setting the initial conditions to zero, of course, since the position and velocity imparted by the forcing function at time zero are not necessarily zero. What we must do is solve the differential equation at time $t = t_0$, and then substitute into the relation the values of the forcing and the forced response at that time, to solve for the initial position and velocity required to match the forced response. Recall our earlier analysis, showing that $x\left(t\right) = x_p\left(t\right) + x_h\left(t\right)$, where:

$$x_h(t) = A_h e^{-\zeta\omega t}\sin(\omega_d t + \theta), \tag{18.2}$$

and

$$x_p(t) = A_0\left(t\right)\cos(\omega t - \phi), \tag{18.3}$$

where we found that the constants A_0 and ϕ are:

$$A_0 = \frac{A\left(t\right)}{\sqrt{\left(\omega_n^2 - \omega^2\right) + \left(2\zeta\omega_n\omega\right)^2}},\ \phi = \tan^{-1}\frac{2\zeta\omega_n\omega}{\omega_n^2 - \omega^2}. \tag{18.4}$$

We need to specify the initial conditions $x\left(t_0\right)$ and $\dot{x}\left(t_0\right)$ so that we can have $A_h = 0$, eliminating the transient response $x_h\left(t\right)$. This is as straightforward as specifying $x\left(t_0\right) = x_p\left(t_0\right)$ and $\dot{x}\left(t_0\right) = \dot{x}_p\left(t_0\right)$, or:

$$\begin{aligned} x\left(t_0\right) &= A_0\left(t_0\right)\cos\left(\omega t_0 - \phi\right) \\ \dot{x}\left(t_0\right) &= -\omega A_0\left(t_0\right)\sin\left(\omega t_0 - \phi\right). \end{aligned} \tag{18.5}$$

This is easily coded into Matlab, using the initial value of the force amplitude in the random array A. We neglect the time derivative of the randomly-varying amplitude A as a first approximation. Figure 18.1 is the displacement versus time curve for the oscillation. The oscillation remains periodic even with the random forcing amplitude, albeit with an irregular amplitude. The force amplitude is shown in Figure 18.2. The plot only shows the first five seconds to give a flavor for the variations in the force.

```
function [t,x,A]=randamp(tspan,wn,zeta,distflag,omega);
% Program 1: randamp.m
% This program solves a single degree
```

[5] This is a coarse approximation, to be sure, but has the benefit of being simple to implement. The reader is welcome to devise other refinements.

```
% of freedom oscillator given an
% input of the form F(t)=Acos(omega*t+phi),
% where A is a random variable.
% distflag should be zero for uniform,
% and 1 for Gaussian distribution.
%
global A_g tspan_g zeta_g wn_g omega_g phi_g % needed by the ode
definition.
if nargin<1 tspan=linspace(0,30,1000); end
if nargin<2 wn=1; end
if nargin<3 zeta=0.05; end
if nargin<4 distflag=0; end
if nargin<5 omega=3.5; end
%
% Assigning global values.
%
tspan_g=tspan; zeta_g=zeta;
wn_g=wn; omega_g=omega; phi_g=0; % Zero phase angle.
% Since we've got to have an amplitude for each
% time step (ease of interpolation), we need to be specific
% about how many time steps we've been given. We'll say that if
% tspan has fewer than 1000 elements, then we make
% it a 1000-element vector.
m=max(size(tspan));
if m<1000
 tspan=linspace(t(1),t(m),1000);
end
%
% Now, we define the amplitude A, between 0 and 5 for
% the uniform distribution, and as a distribution with
% mean of 2.5 for the Gaussian. We will force A>=0,
% so the Gaussian distribution will need a little
% bit of tweaking (and it won't be rigorously Gaussian).
%
switch distflag
 case 0
 A=5*rand(size(tspan)); % A now has a range from 0 to 5.
 case 1
 A=2.5+randn(size(tspan)); % One standard deviation will span [1.5,3.5].
 A=A.*sign(A); % Multiplying A by the signum function of itself.
 otherwise
 error('Distribution flag must be zero or one.');
end
%
% Next, we define the initial conditions to remove transients.
%
```

```
phi=atan(2*zeta*omega*wn/(wn^2-omega^2));
x0=A(1)*cos(omega*tspan(1)-phi);
xd0=-A(1)*omega*sin(omega*tspan(1)-phi);
xv0=[x0,xd0];
%
% Storing A in the global value.
%
A_g=A;
%
% Now we're ready to solve the ODE.
%
[t,x]=ode45('randdx',tspan,xv0);
function dx=randdx(t,x)
%
% Program 2: randdx.m
% Differential equation definition for
% the randomly-varied amplitude case.
%
global A_g tspan_g zeta_g wn_g omega_g phi_g % Need these for the
interpolation.
At=interp1(tspan_g,A_g,t); % Finding the correct value of A.
% interp1 is a built-in function.
%
dx(1,1)=x(2,1);
dx(2,1)=At*cos(omega*t-phi)-(2*zeta*wn*x(2,1)+wn^2*x(1,1));
```

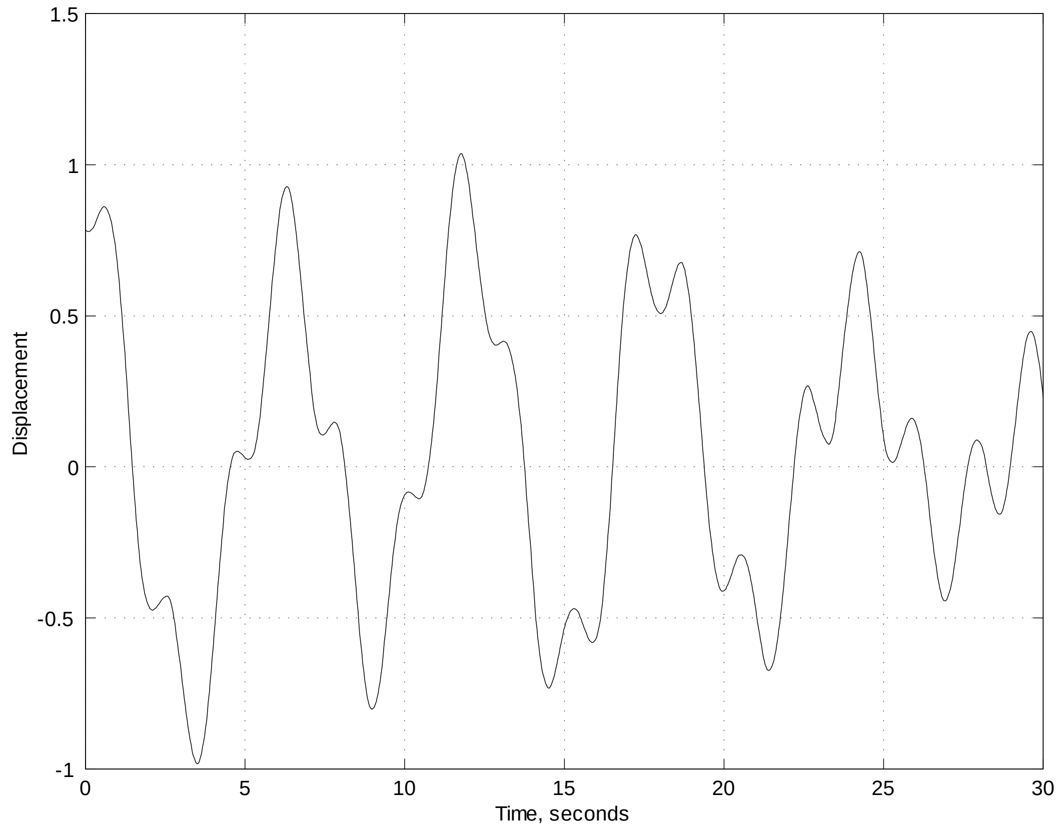

Figure 18.1: Displacement versus time plot for the random Duffing oscillator.

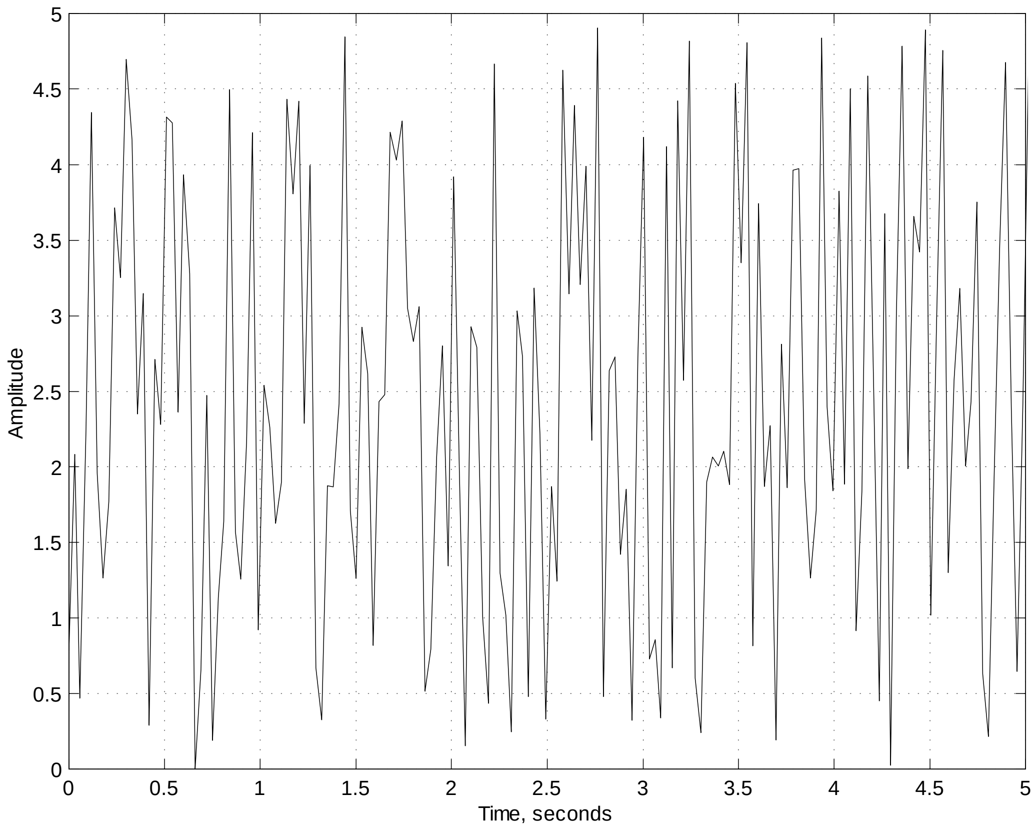

Figure 18.2: Amplitude versus time curve for five seconds, to show variations.

Chapter 19

Randomly-Excited Duffing Oscillator

This example expands on some of the ideas of the previous, in that now we will apply a random forcing input to a nonlinear oscillator. We could call upon the van der Pol oscillator of a few examples ago, but instead we will introduce the Duffing equation:

$$\ddot{x} + c\dot{x} + kx + \varepsilon g\left(x\right) = A \cos \omega t \tag{19.1}$$

where the parameters c and k are assumed to be positive, and $|\varepsilon| \ll 1$. Note that ε is not restricted to solely positive nor negative values. This equation allows us to choose the parameter ε and the function $g\left(x\right)$ to model nearly-linear springs, for example. Also, this equation retains some attractive quasi-linear qualities. Specifically, if we set $A = 0$ and $c = 0$, we would expect roughly oscillatory motion for small amplitudes x, and that if we further introduce a small damping coefficient c, these small-amplitude oscillations would reduce to zero. If we assume a function $g\left(x\right)$ such that $\operatorname{sgn}(g\left(x\right)) = \operatorname{sgn}(x)$, then the sign of the parameter ε determines the character of the stiffness element being modeled.[1] If $\varepsilon < 0$, then we see that the restoring force will be smaller in extension, and arrive at a *soft* spring. Conversely, if we take $\varepsilon > 0$, the spring gains stiffness in extension over the purely linear case, and is called a *hard* spring.

Of course, we are not limited to the class of functions $g\left(x\right)$ described above. For example, if we choose the parameter $\varepsilon > 0$ and select $g\left(x\right) = -x^2$, we then get a net restoring force $kx + \varepsilon g\left(x\right)$ that is negative for $x < 0$ and positive for $x > 0$. In other words, the spring will be soft in extension but hard in compression, meaning that the center of oscillation (equilibrium point) will be shifted slightly away from zero, where the magnitude of the shift depends on the relative values of k and ε.

[1]Recall from an earlier example that $\operatorname{sgn}(x)$ is the signum function, which returns -1 if x is negative, 1 if x is positive, and 0 if x is zero.

Let us use the Duffing oscillator as a model for response to a harmonic input. Programs 1 and 2 below use the differential equation solver suite inside Matlab to calculate a numerical solution to a Duffing equation, given the input values of c, k, ε, A, and ω. The function uses $g(x) = -x^2$; this can be readily changed by editing the differential equation definition function (Program 2). Also, the time span and initial conditions are fixed inside the code, to keep the number of inputs manageable. Program 1 can be quickly edited to take a different time increment or initial conditions, or to take them as function inputs. Figure shows the results of the hard-coded default case.

Having arrived at a solution for the Duffing equation, our next step is to introduce a random forcing function. Unlike the previous example (and in preparation for the next) we will select a random forcing frequency and amplitude, which will remain constant for the duration of the oscillation. If we design this function properly, we will not need to write a new program to provide the differential equation; we will send the same set of global variables. This implementation is given in Program 3 below. Again, we leave the time interval and initial conditions defined within the function. (The reader is encouraged to rewrite the function to take the time increment, initial conditions, or both as inputs, providing default values via `nargin`.) We use the uniform distribution generator inside Matlab to provide a forcing amplitude between zero and 5, and a forcing frequency between zero and 2π. Note that the function returns four arrays: the time vector t, the state vector x, the random amplitude A, and the random frequency w. This is for ease of comparison between runs, and also makes for a useful check on our random solution. Knowing the values of A and w used by the code, we can run our deterministic solver (Program 1) using the default values for e, c and k, and the random values for A and w. The results for a few default runs are plotted below, along with the random frequency and amplitude used. Of special note is the third run in Figure 19.3. It happened that we got lucky (or unlucky, depending on the point of view) on this run, and got a random forcing frequency very close to the system's natural frequency of one. Hence, we see a nearly-resonant condition in this undamped oscillator. The phase diagram in Figure 19.4 corresponds to this run. The response traces out arcs in the phase plane that are circular, but by no means closed. Examining the plot point by point shows that the oscillation travels back and forth through the phase plane, meaning that the mean point of the oscillation does not move one way or the other; it is stationary at zero. The interested reader is encouraged to look at this behavior for himself. A simple way to accomplish this is to run the Duffing oscillator function with the given parameters, and then type the following at the Matlab prompt:

```
for i=1:max(size(x))
plot(x(i,1),x(i,2),'o')
hold on
pause
end
```

Matlab will execute this loop as if it was inside a function or script, and you will be able to track the movement in the phase plane.[2]

```
function [t,x]=simpduff(e,c,k,A,w)
% Program 1: simpduff.m
% Function to set up and implement the ode45
% solution of a Duffing oscillator. The
% time increment is hard-coded to be
% [0,30], and the initial conditions
% set to [0,0].
%
global c_g k_g e_g A_g w_g % Will be used in ode solution.
tspan=[0 30];
xinit=[0 0]';
% The customary nargin check.
if nargin<1 e=0.01; end
if nargin<2 c=0.05; end
if nargin<3 k=1; end
if nargin<4 A=3; end
if nargin<5 w=2.7; end % w chosen to prevent resonance.
[t,x]=ode45('duffdx',tspan,xinit);
function dx=duffdx(t,x)
% Program 2: duffdx.m
% Function to return the updated state
% vector for a Duffing oscillator.
%
global c_g k_g e_g A_g w_g
g=-x(1,1)^2; % Explicitly stating g(x).
dx(1,1)=x(2,1);
dx(2,1)=A_g*cos(w_g*t)-(c_g*x(2,1)+k_g*x(1,1)+e_g*g);
% Since we stated g(x) above, the equation for x”
% looks like the Duffing equation. Also, we need
% only change that one line of code above to use
% a different function g(x).
function [t,x,A,w]=randduff(e,c,k)
%
% Program 3: randduff.m
% This function provides random force amplitude
% and frequency to a Duffing oscillator. Both values
% are taken from a uniform distribution.
%
global e_g c_g k_g A_g w_g % Tagging these with “_g” for consistency.
tspan=[0 30];
xinit=[0 0]';
if nargin<1 e=0.01; end
if nargin<2 c=0.05; end
```

[2]You may also have to sit through many pauses; the x array we computed was 2-by-457.

```
if nargin<3 k=1; end
rand('state',sum(clock*100)); % Resetting random generator state.
A=5*rand(1);
w=2*pi*rand(1);
e_g=e; c_g=c; k_g=k; A_g=A; w_g=w;
[t,x]=ode45('duffdx',tspan,xinit);
```

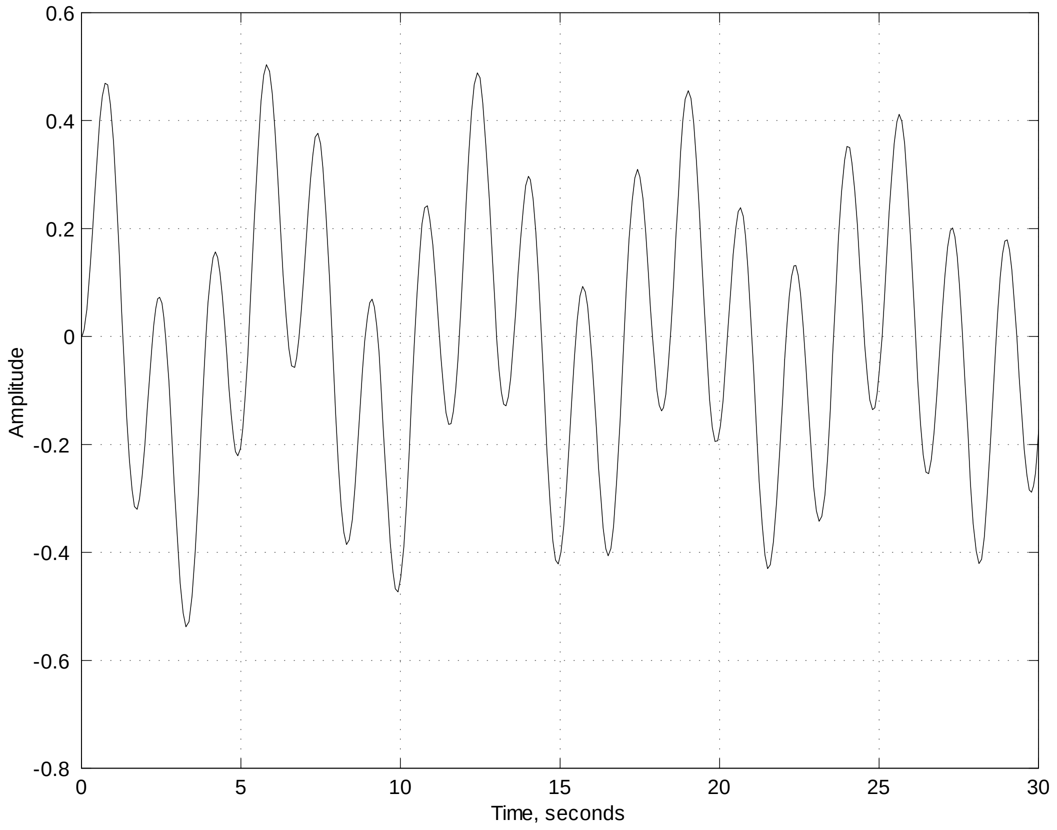

Figure 19.1: Response of the randomly-excited Duffing oscillator with $\omega = 3.7960$ and $A = 3.7999$.

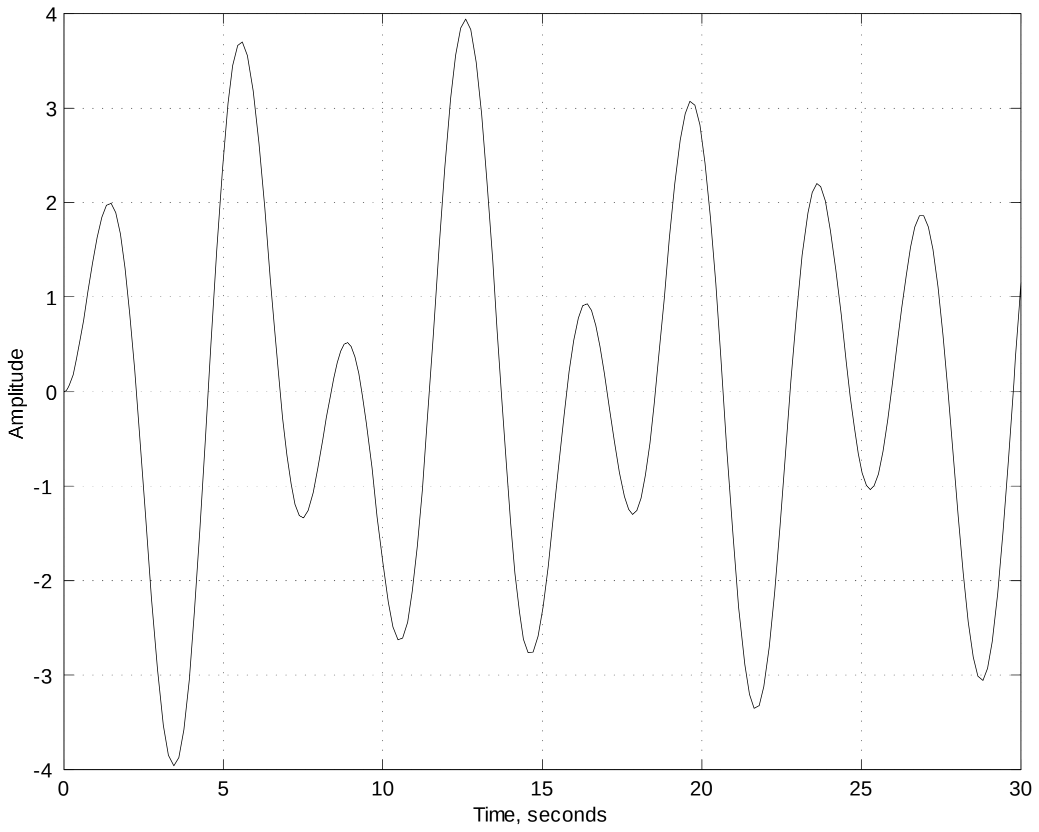

Figure 19.2: Response of the Duffing oscillator to amplitude $A = 4.4531$ and forcing frequency $\omega = 1.7404$.

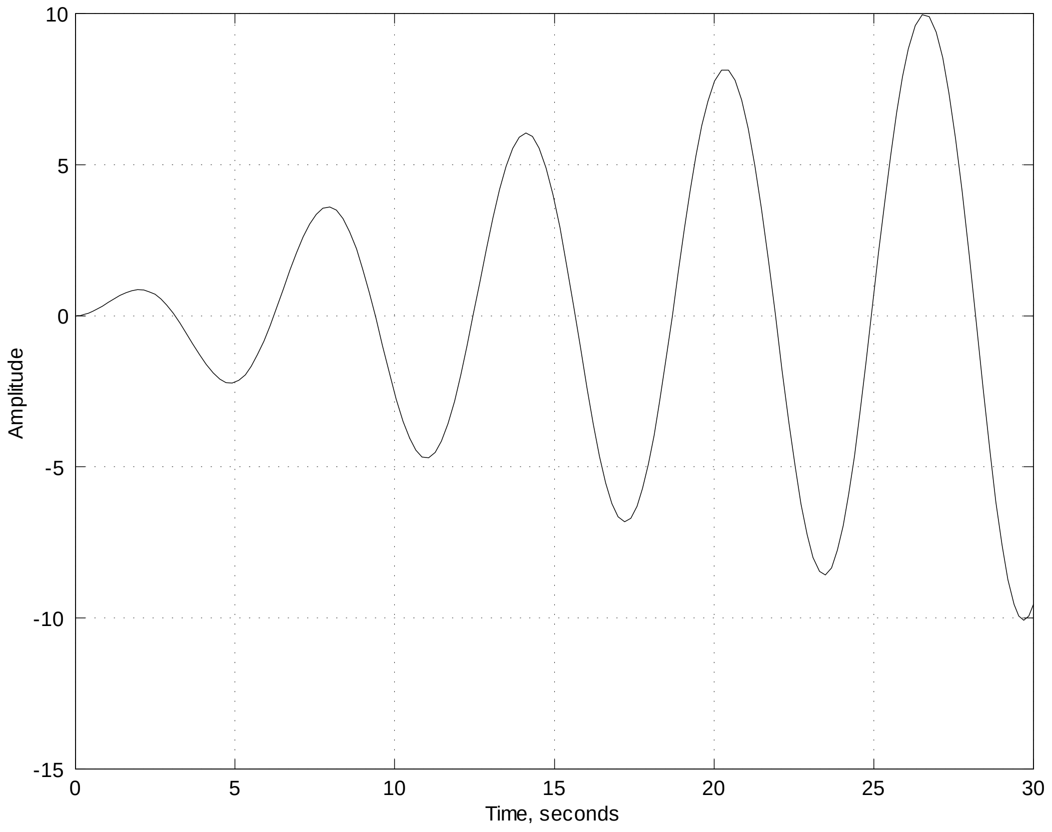

Figure 19.3: Response to $A = 1.0062$ and $\omega = 1.0115$.

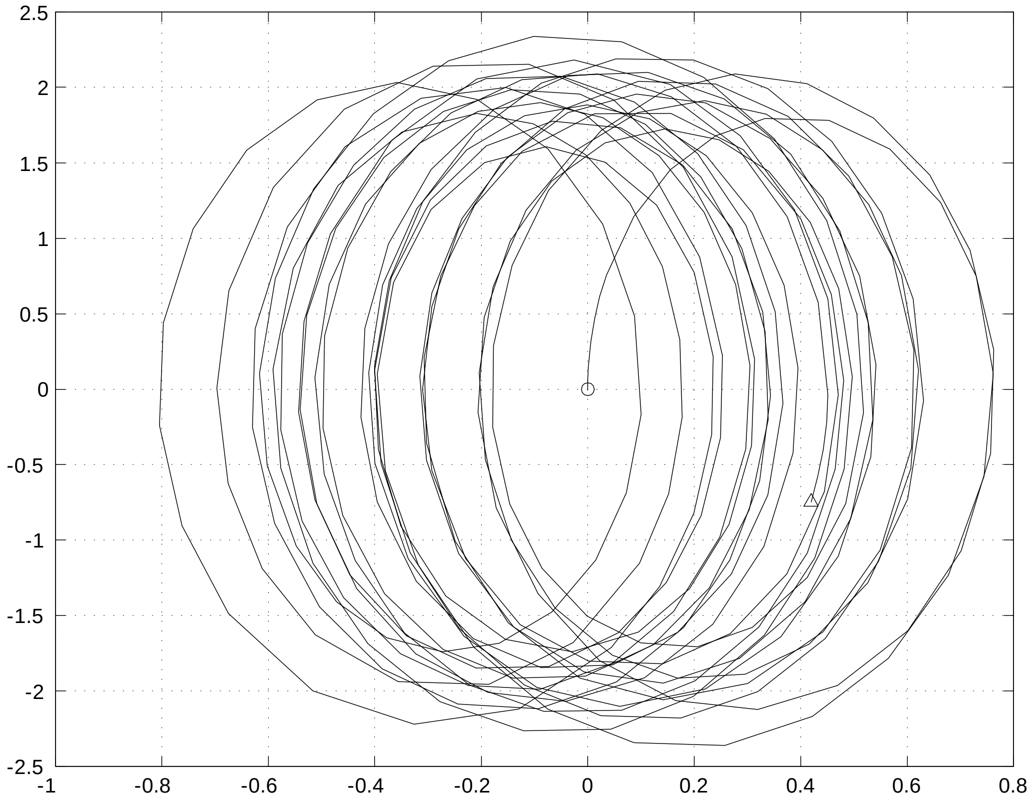

Figure 19.4: Phase plot for the nearly-resonant case above.

Chapter 20

Monte Carlo Simulation of a Random System

In the previous example, we looked at a system under random excitation, where we modeled the amplitude of a harmonic input force as a time-varying random process, using a normal or uniform distribution. Consider now a related problem, where we again have a harmonically-excited single degree of freedom oscillating system, where the excitation is of the form:

$$F(t) = A\cos(\omega t), \tag{20.1}$$

where the parameters A and ω are random variables. As with the previous analysis, we assume that the forcing will be applied in a certain range, that is:

$$\begin{aligned} A_1 &\leq A \leq A_2, \\ \omega_1 &\leq \omega \leq \omega_2 \end{aligned} \tag{20.2}$$

where A_1, A_2, ω_1, ω_2 are known constants. One approach would be to model the results with A and ω at mean values, which could form a good basis to estimate the behavior of a linear system. However, let us address the more interesting (and difficult) problem of estimating the behavior of a nonlinear system, such as the Duffing oscillator from the previous example. There is no longer a guarantee that the mean values for the inputs will correspond to the mean values of the response, so we need an alternate method to arrive at some response statistics.[1]

To approximate the statistics of our nonlinear system, we choose to employ the Monte Carlo methodology, developed systematically by Metropolis and Ulam (1949). In general, the Monte Carlo method has two forms: the probabilistic form, where actual random variable distributions are available, and the

[1]In addition to the numerical approach presented here, one can apply perturbation theory to a nonlinear system to estimate the response. For more, see *Nonlinear Ordinary Differential Equations*, by D.W. Jordan and P. Smith (Oxford University Press), or *A First Look at Perturbation Theory,* by J. G. Simmonds and J. E. Mann, Jr. (Dover Books).

deterministic form, where such distributions are unknown. Here, we follow a deterministic algorithm, though adaptation of the method as described here to its probabilistic form is a matter of substituting the known random distributions for the ones we will presume here.

The method in both of its forms can be broken down into three steps:

1. Simulation of the random variable(s);

2. Solution of the deterministic problem for a large number of random variable realizations;

3. Statistical analysis of results.

We have already performed most of the work for the first step above. Since we have ranges over which we presume the random variables A and ω will vary, we can describe both as uniformly-distributed between their extreme values. We could also use a Gaussian distribution, using for the mean the midpoint of the given ranges. Our standard deviation would then be selected so the endpoint values given correspond to three standard deviations from the mean value in either direction. In this way, about 99.7% of the randomly-generated values would fall between the prescribed extreme values. We will demonstrate the procedure for uniformly-distributed variables, and leave the Gaussian distribution as an exercise for the reader.

In order to solve the deterministic problem, we will use the differential equations definition function we presented in the previous example. What we need to create is a separate driver for this function, a code that will run several realizations and return our desired quantities. Of course, this means that we need to determine which output quantities interest us. For the purpose of demonstration, we will track the maximum displacement and maximum velocity for each realization. If it was necessary, we could retain the entire history of oscillation for each realization, but a simple demonstration will capture the spirit of the Monte Carlo methodology. The reader is encouraged to modify the given code to add whatever complication he wishes.[2]

We will only arrive at useful statistics for our system if we can apply the conditions of ergodicity to the system. Thus, we will again define the system initial conditions so that the transients are removed, like we did with our first example of a random forcing input. These values can either be computed as needed, right before sending to the ODE solver, or they can be computed in advance. We choose the former for our demonstration. (The reader may be wondering if this calculation can be vectorized. It can; the programming is left as an exercise.) Also, with the manner in which we have chosen to specify global variables, we can set the value of the global amplitude and frequency right before calling for the ODE solution. We specify the random variables as row vectors (arbitrarily) and we force them to lie on the desired domains. Then, we set up

[2]A word of warning, however: Reduce the value of the damping coefficient c at your own risk. Our initial value for this parameter was 0.05, and the integrator did not complete any attempted run of the code at that value of c.

a `for` loop to run the ODE solver the specified number of times, solving for the appropriate initial displacement and velocity for each. After the ode solution is found, the maximum displacement and velocity is found and stored, as are the input amplitude and frequency, and the time span used. We may then perform whatever statistical analyses we wish on this data.

The four histograms below show the variations in several parameters for a set of 100 random realizations of the Duffing oscillator. Figure 20.1 is the histogram of force amplitudes, and Figure 20.2 is the set of forcing frequencies. Since we chose uniform distributions for both of these quantities, we expect the number of values in each range to be about the same. This is the case; the difference between the most and least common force amplitude range is eight occurrences, and the difference between the most and least common frequency range is ten occurrences. For a larger sample size, we would expect these numbers to reduce on a percentage basis.[3]

Our results, however, are not uniformly distributed. Figure 20.3 shows the variation of maximum displacement values. The vast majority (65%) had maximum displacement under 10. The shape of the distribution is akin to a one-sided Gaussian distribution, implying that our normally-distributed inputs led to Gaussian output. Figure 20.4 also shows a one-sided Gaussian distribution, with 85% of the resulting maximum velocities falling under 10. One can readily see how this information could be applied to a design procedure; if we wanted to have 90% certainty of a displacement less than 10, we would see that the default system is inadequate.

It is also possible to do more complicated analyses involving other aspects of the data. The reader is invited to rewrite the routine to return all of the displacement and velocity histories (hint: three-dimensional matrices will be helpful). Also, if we had a nonlinear model for, say, a vibrating beam, we could use the displacement and velocity information to derive the maximum stress at some point in the beam for each run. The possible applications of this analysis are limited only by the systems we wish to apply it to.

```
function [xout,xdout,Aout,wout]=montduff(nruns,e,tspan)
%
% Program 1: montduff.m
% This code performs a Monte Carlo simulation of the
% Duffing oscillator. The random parameters are the
% forcing amplitude and frequency, presumed to be
% described by uniform distributions. Also, the
% stiffness is presumed to be one and the damping
% presumed to be 0.1. These values can be changed
% by editing the code.
%
global c_g k_g A_g w_g e_g % For compatibility with the existing duffdx.m.
```

[3] A test performed with 1,000 simulations showed that the difference in amplitude occurences was 30 of 1000 between most and least common, or 3%. The difference in number of occurences for the most and least common frequency was also around 30 out of 1000.

```
% Parameter check.
%
if nargin<1 nruns=50; end
if nargin<2 e=0.01; end
if nargin<3 tspan=[0 30]; end
%
% Stiffness and damping coefficient.
%
c_g=0.1; k_g=1; e_g=e;
wlo=0; whi=2*pi; % Extreme values for the frequency.
Alo=0.1; Ahi=10; % Extreme values for the amplitude.
A=Alo+(Ahi-Alo)*rand(1,nruns);
w=wlo+(whi-wlo)*rand(1,nruns);
%
% Initializing the t and x vectors. This is why we want
% to specify the length of the time vector.
%
xout=zeros(1,nruns);
xdout=zeros(1,nruns);
for i=1:nruns
 A_g=A(1,i); w_g=w(1,i); % Set global parameters.
 x0=A_g*cos(w_g*tspan(1));
 xd0=-A_g*w_g*sin(w_g*tspan(1)); % Choose initial conditions to remove
transient.
 xinit=[x0, xd0];
 [trecv,xrecv]=ode45('duffdx',tspan,xinit); % The run.
 xout(1,i)=max(xrecv(:,1)); % Here we could perform a more complex analy-
sis
 xdout(1,i)=max(xrecv(:,2)); % of our data.
end
wout=w;
Aout=A;
```

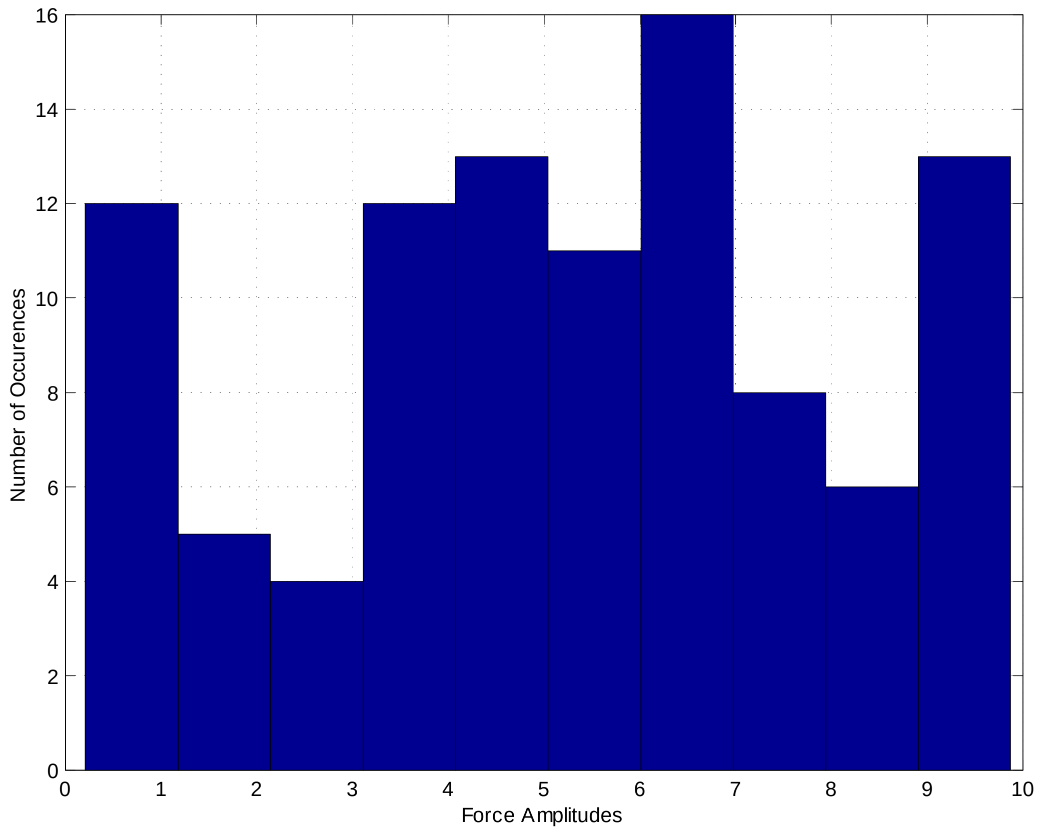

Figure 20.1: Histogram showing the force magnitudes among the 100 random runs.

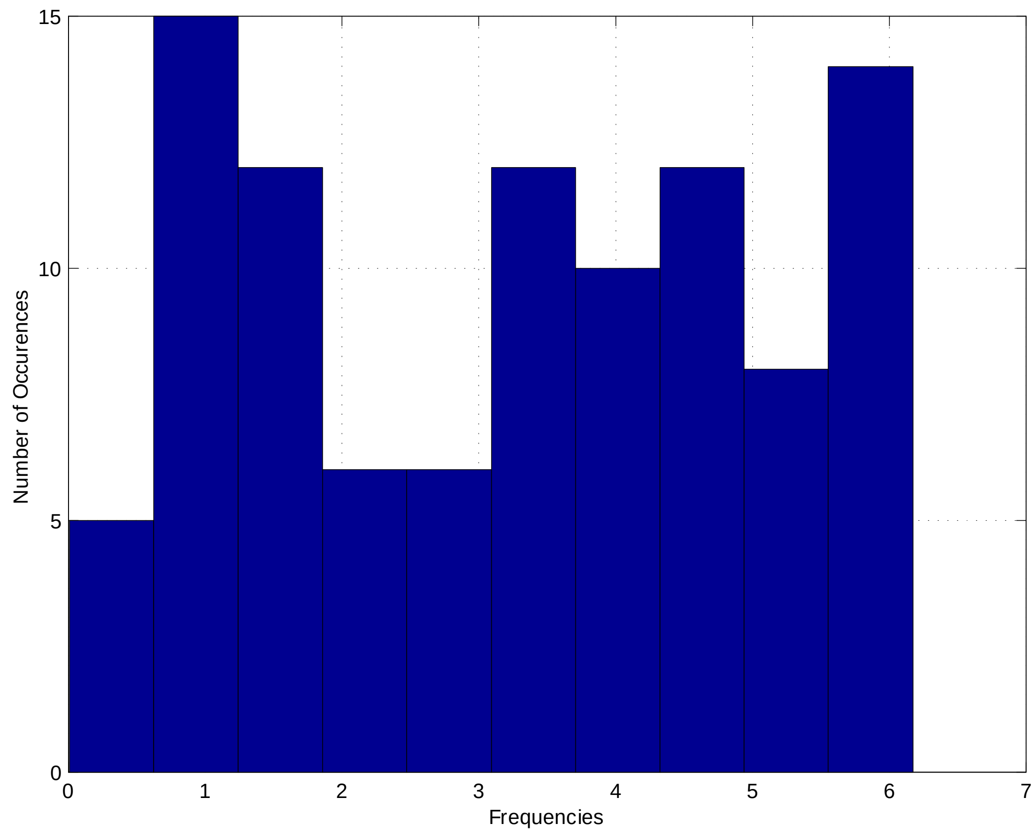

Figure 20.2: Histogram showing the various frequencies applied to the Duffing oscillator.

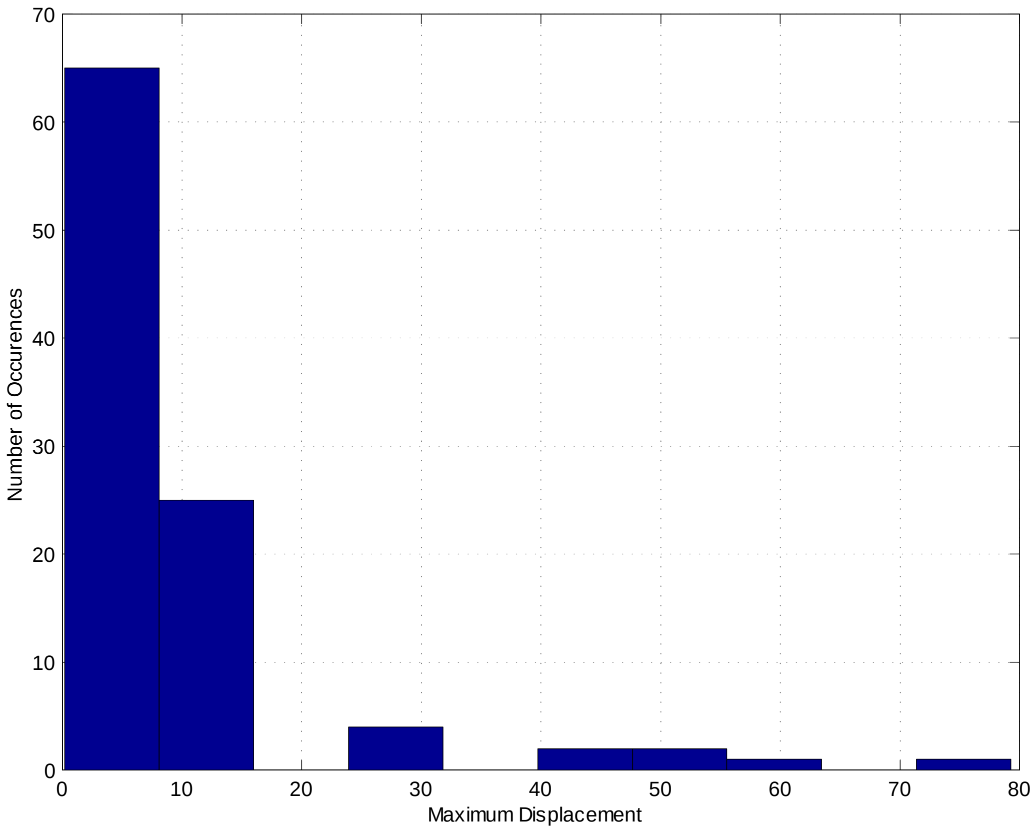

Figure 20.3: Histogram of maximum displacement values.

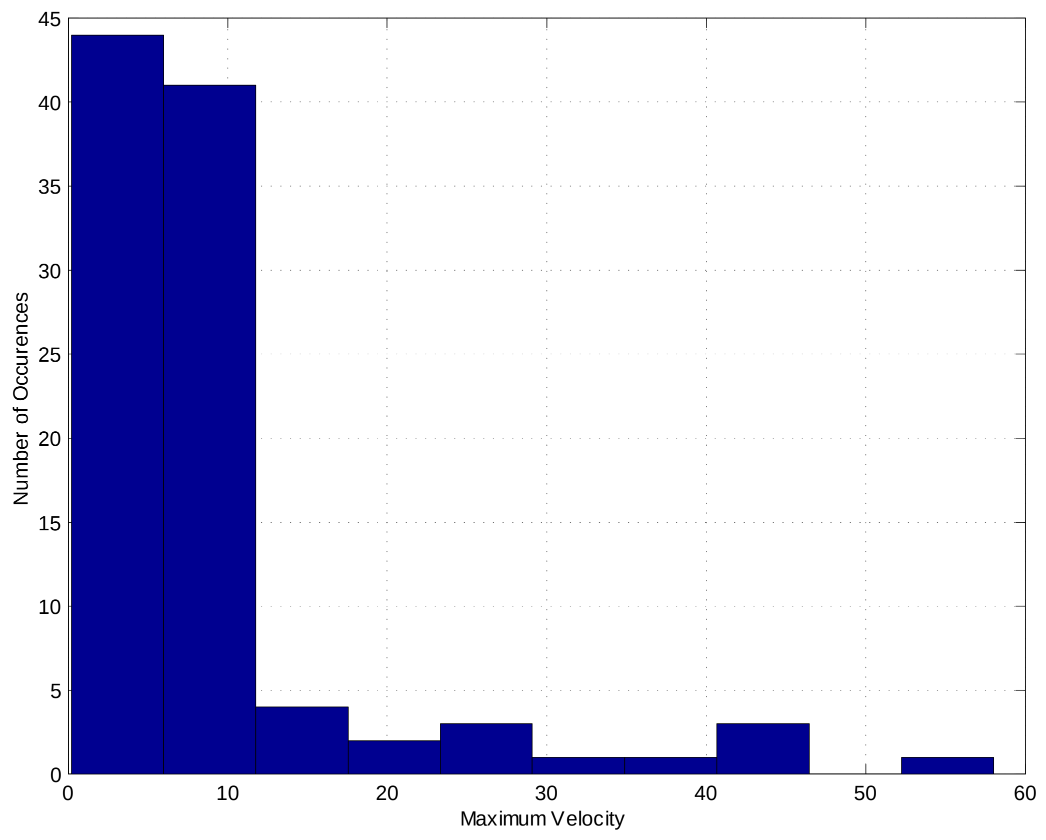

Figure 20.4: Histogram showing maximum velocities of the simulated responses.

Chapter 21

Conclusion

This primer on MATLAB as applied to vibration problems, simple and not so simple, can be an valuable tool to the student and the practitioner. The basic elements are here to study a variety of vibration problems and to “play” with the equations and the parameters. You will find it interesting and fun doing this. Best of luck.