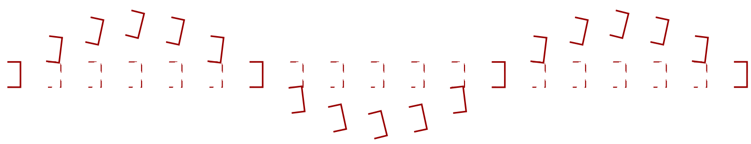
Asger Juul Brunshøj

Visualization of

Beam with Coupled Bending and Torsion Vibrations

Bachelor's project



June 2014

DTU Civil Engineering
Department of Civil Engineering

Abstract

Beams with cross sections that are not doubly symmetric exhibits coupled bending and torsion vibrations. The governing equations of motion for the coupled vibrations are derived from equilibrium equations. The equations are solved with a series expansion, using the method of assumed modes to derive basis functions from the uncoupled equations of motion. Solutions are implemented in MATLAB by building a program featuring a graphical user interface, allowing students to get a feel for coupled vibrations by visualizing the vibrations for different settings and parameters.

I would like to thank my advisor, Jan Becker Høgsberg, for giving me the idea for this project, as well as for his guidance.

Contents

List of Figures

List of Tables

Notation

$\mathbf{0}$	A vector or matrix comprised of zeros, depending on context.
A	Cross-sectional area.
A_k	A constant of integration in the time response function of the k'th mode shape.
$\mathbf{b}$	A vector introduced when linearizing the system of ODEs.
B_k	A constant of integration in the time response function of the k'th mode shape.
c	Distance between the center of mass and the shear center.
C	Location of the center of mass.
C_i	Arbitrary constants of integration for $i = 1, 2, \ldots$.
dx	Infinitesimal length of beam segment.
D_n	A constant factor in $W_n(x)$.
E	Elasticity modulus.
$\mathbf{f}$	A vector consisting of elements relating to the external force $p(x,t)$.
F_n	A constant factor in $\Phi_n(x)$.
G	Shear modulus.
$\mathbf{H}$	A matrix introduced when linearizing the system of ODEs.
i	Imaginary unit.
$\mathbf{I}$	The identity matrix of varying size depending on context.
I_{CM}	Moment of inertia of beam segment about an axis going through its center of mass.
I_m	Moment of inertia of beam segment about an axis going through its shear center.
I_p	Polar moment of area defined by $I_p = I_y + I_z = \int_A y^2 \, \mathrm{d}A$.
I_y	Second moment of area of the cross section defined by $I_y = \int_A y^2 \, \mathrm{d}A$, where y is the distance to the center of mass along the y axis.
I_z	Second moment of area of the cross section defined by $I_z = \int_A z^2 \, \mathrm{d}A$, where z is the distance to the center of mass along the z axis.
K	Torsion constant.
$\mathbf{K}$	Stiffness matrix.
L	Beam length.
$M(x,t)$	Internal moment.
$\mathbf{M}$	Mass matrix.
O	Location of the shear center. The shear center lies on the line $(x,0,0)$.
$p(x,t)$	External force.
$\mathbf{q}$	The vector $\left[[\mathbf{z} \quad \mathbf{y}]^\top\right]$ introduced when linearizing the system of ODEs.
$Q(x,t)$	Internal shear force.
$r_n(t)$	A temporal part of the series expansion for deflection due to bending $w(x,t)$.
$R_k(t)$	A definite integral combining bending basis functions $W_k(x)$ with the external force $p(x,t)$.
$s_n(t)$	A temporal part of the series expansion for angular deflection due to torsion $\phi(x,t)$.
$S_k(t)$	A definite integral combining torsion basis functions $\Phi_k(x)$ with the external force $p(x,t)$.
$\mathbf{v}_k$	The k'th eigenvector.

$w(x,t)$	Deflection due to bending.
$W_n(x)$	A basis function in the series expansion for bending.
$\mathbf{y}$	A time derivative of $\mathbf{z}$ introduced to linearize the system.
$\mathbf{z}$	A vector of time response functions.
δ_{kn}	Kronecker-delta defined as $\delta_{kn} = 1$ if $n = k$ or $\delta_{kn} = 0$ if $n \neq k$.
ω_k	The k'th natural frequency.
$\frac{\partial}{\partial x}$	Partial derivate with respect to x. This will sometimes be written with an apostrophe as in $w'(x,t)$.
$\frac{\partial}{\partial t}$	Partial derivate with respect to t. This will sometimes be written with a dot as in $\dot{w}(x,t)$.
$\phi(x,t)$	Angular deflection due to torsion.
$\Phi_n(x)$	A basis function in the series expansion for torsion.
$\psi_{k,n}$	A definite integral combining bending and torsion basis functions $W_k(x)$ and $\Phi_k(x)$.
$\mathbf{\Psi}$	A matrix made from elements $\psi_{k,n}$.
ρ	Density.
$\tau(x,t)$	Internal torque.

1
Introduction

A beam with a doubly symmetric cross section, like an I-profile ⌶ , will vibrate in pure bending when subjected to an external load or bending moment. It may also vibrate in pure torsion, as is the case when it is subjected only to an external torque. On the other hand, beams with cross sections featuring only a single axis of symmetry (or none at all) will be subject to coupled bending and torsion even when subjected to only an external load, or only a torque, since the coupling comes from the inertia of the beam as it vibrates. An example of a beam with only a single axis of symmetry is a C-clamp profile] . The coupling is caused by the shear center and the center of mass not coinciding. A homogenous doubly symmetric cross section will always have coinciding shear center and center of mass, but the same is not true for cross sections with only a single or no axis of symmetry. Figure 1.1 attempts to illustrate the motion of an ⌶ cross section with two axes of symmetry contrasted by a] cross section with a single axis of symmetry.

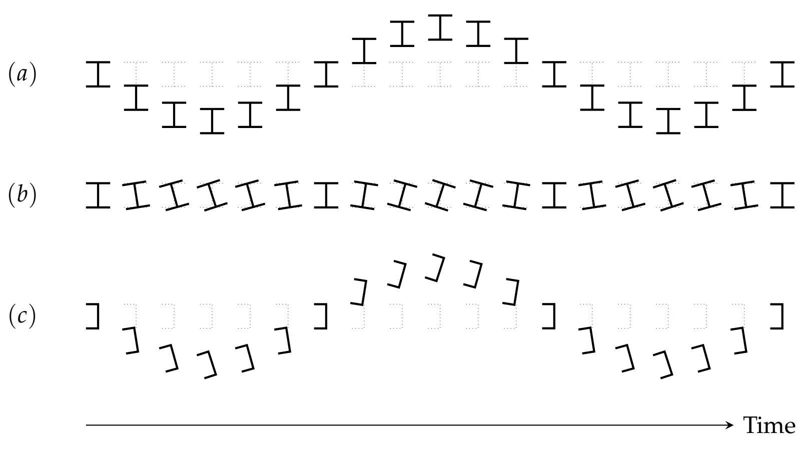


Figure 1.1: Conceptual drawing of pure bending or torsion vibrations and coupled bending and torsion vibrations.
(*a*): A doubly symmetric cross section of a beam with an I-profile in pure bending. (This could be a typical bernouille–euler beam).
(*b*): A doubly symmetric cross section of a beam with an I-profile in pure torsion.
(*c*): A C-clamp profile cross section with a single axis of symmetry exhibiting coupled bending and torsion vibrations.

The ambition of this project is to build a tool in MATLAB to visualize coupled bending and torsion vibrations. This program is intended for use by students. The program is built around a graphical user interface (GUI) that allows the user to easily change parameters and play with various settings. As the equations are complicated enough as is, we shall limit ourselves to investigating coupled bending and torsion vibrations for uniform beams with homogenous cross sections with a single axis of symmetry. This has the positive effect of making the software easier to use and allows it to serve as a good introduction to coupled bending and torsion vibrations.

The content of chapter 2 is the derivation of the governing equations of motion for the coupled vibrations. Chapter 3 focuses on solving these equations, and the implementation in MATLAB is discussed in chapter 4.

2

Deriving the Coupled Equations of Motion

Figure 2.1 shows a beam with a C-clamp profile. It will experience coupled bending and torsion vibrations since it only has a single axis of symmetry. The axis of is about the y axis. This chapter derives coupled equations of motion from equilibrium equations for a beam of this kind.

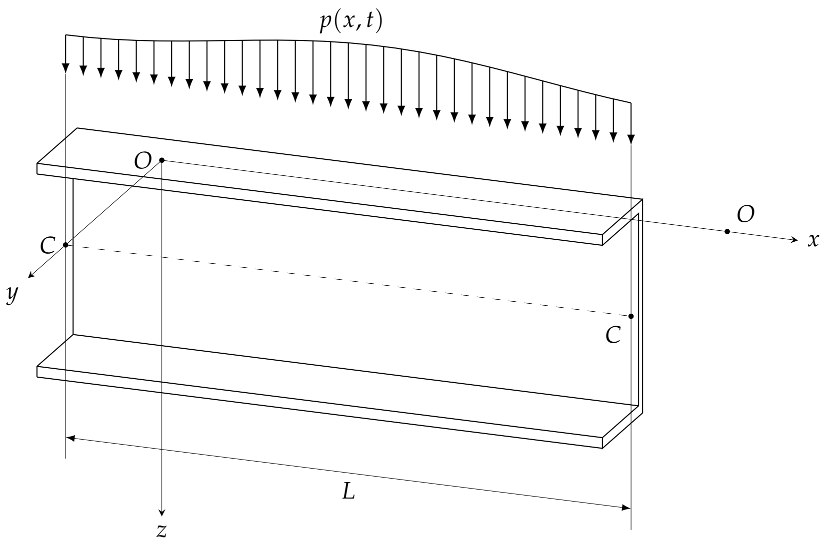


Figure 2.1: A uniform beam of length L subjected to a load $p(x,t)$. Here, O indicates the location of the shear center and C indicates the location of the center of mass. The coordinate system is placed with origin at the shear center, at the beam end.

The beam is uniform with length L. It may be subject to a load $p(x,t)$, which acts in the z direction. We'll let O denote the shear center, and C the center of mass. A coordinate system (x, y, z) is placed with its origin at the shear center at one end.

A beam segment of infinitesimal length dx is shown on Figure 2.2. This includes relevant internal forces and moments, where Q denotes shear forces, M is a moment about the y axis and τ denotes a torque about the x axis. These are all functions of space x and of time t, $Q(x,t)$, $M(x,t)$ and $\tau(x,t)$, but are written as Q, M and τ as short notation. The same is true for p and $p(x,t)$ in the following. p acts on a line which goes through the center of mass C.

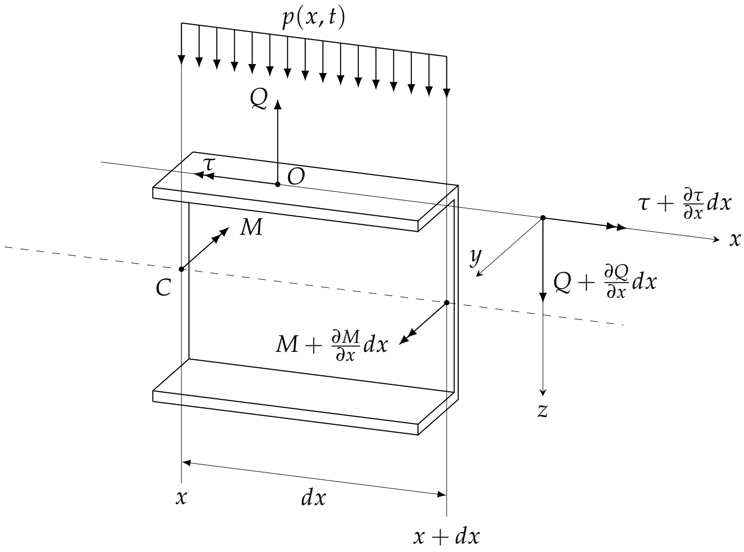


Figure 2.2: A beam segment of infinitesimal length dx, with relevant cross-sectionalal forces and moments.

2.1 Equilibrium of moments

Equilibrium of moments about the y axis, when taken at the right side of the beam segment at $x + dx$, gives

$$M + M' dx - M + p\frac{dx^2}{2} - Q\,dx = 0. \tag{2.1}$$

The right hand side is zero, since the rotary inertia is assumed to be negligible. The dx^2 term is vanishingly small, and upon division by dx and rearranging, this becomes

$$M' = Q. \tag{2.2}$$

Note in the following that an apostrophe, as in M', will be used as short notation for partial derivatives with respect to x. The short notation will be used interchangably with the full notation $\frac{\partial M}{\partial x}$, depending on whether the given context favors brevity or clarity of structure. Likewise, a dot shall be used to denote partial derivates with respect to time t, as in $\dot{M}$, and will be used interchangeably with $\frac{\partial M}{\partial t}$. The full notation is usually preferred in favor of clarity when introducing a partial derivative spanning multiple terms.

2.2 Force equilibrium

Force equilibrium in the direction of the z axis gives

$$Q + Q' dx - Q + p\,dx = -\rho A\,dx\,\frac{\partial^2}{\partial t^2}\left(w - c\phi\right), \tag{2.3}$$

where ρ is the density, A is the cross-sectional area, w is short notation for the deflection due to bending $w(x,t)$, and ϕ is short notation for the angular deflection $\phi(x,t)$, see Figure 2.3. ϕ represents an angle in radians. w is defined positive in the upwards direction, and ϕ is defined positive counterclockwise. c is the distance between the shear center and the center of mass. The right hand side of equation (2.3) is the cross-sectional mass, times its downward acceleration. Here small angles of twist are assumed, in order to obtain a linearized measure for the downward displacement of the center of mass, which then becomes $-(w - c\phi)$.

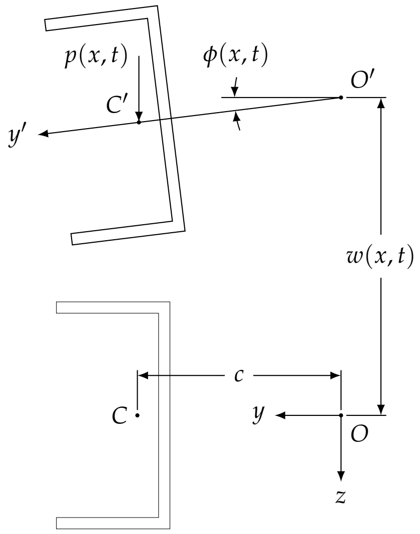


Figure 2.3: This defines the positive direction of deflection due to bending w, and angular deflection due to torsion ϕ. O' and C' are the shear center and center of mass respectively, of the deflected cross-section. (The distance between the shear center and the center of mass has been exaggerated in this figure. This is not the true position of the shear center for this cross section.)

Division by dx and substituting with equation (2.2) leads to

$$M'' + p = -\rho A (\ddot{w} - c\ddot{\phi}) . \tag{2.4}$$

Using the relation

$$M = EI_z w'' , \tag{2.5}$$

which is derived in Krenk and Høgsberg (2013)[1], where E is the elastic modulus and I_z is the second moment of area which, for an infinitesimal beam length dx, is computed by

[1] Steen Krenk and Jan Becker Høgsberg. *Statics and Mechanics of Structures*. 2013. ISBN 978-94-007-6112-4

$$I_z = \int_A z^2 \, \mathrm{d}A, \tag{2.6}$$

with z denoting the distance to the elastic center along the z axis. Combining equation (2.4) and (2.5) yields

$$\frac{\partial^2}{\partial x^2} \left(EI_z w'' \right) + p = -\rho A (\ddot{w} - c\ddot{\phi}) . \tag{2.7}$$

Since the beam is assumed uniform, the first term simplifies, and with a bit of rearranging this leaves us with

$$EI_z w'''' + \rho A \ddot{w} = -p + c\rho A \ddot{\phi}, \tag{2.8}$$

which is the governing bending equation. It is coupled with torsional deflections ϕ on account of c being non-zero.

2.3 *Equilibrium of torques*

Equilibrium of torques about an axis parallel to the x axis passing through the moving deflected shear center O', gives

$$\tau + \tau' \, dx - \tau + c\rho A \, dx \, \ddot{w} + cp \, dx = \ddot{\phi} I_m . \tag{2.9}$$

Here, the right hand side is the angular acceleration $\ddot{\phi}$ times the moment of inertia of the beam segment, I_m. This is the moment of inertia of the beam segment about its shear center. The second term on the left hand side stems from an inertial force of magnitude $\rho A\, dx\, \ddot{w}$ acting on the center of mass.

Employing the parallel axis theorem, the moment of inertia I_m may be written in terms of a moment of inertia about an axis going through the center of mass, the centroidal moment of inertia I_{CM}:

$$I_m = I_{CM} + c^2 \rho A\, dx. \tag{2.10}$$

Assuming a homogenous cross-section, I_{CM} may be expressed in terms of the second moment of area, with respect to an axis parallel to the x axis passing through the center of mass. We will call this second moment of area the polar moment of area, I_p, defined as

$$I_p = \int_A y^2 + z^2 \,\mathrm{d}x, \tag{2.11}$$

where y and z denote distances to the center of mass along the y and z axes, respectively. The centroidal moment of inertia I_{CM} can be expressed as

$$I_{CM} = \rho\, dx\, I_p. \tag{2.12}$$

This equality is explained in the appendix, section A.1. Combining equations (2.9), (2.10) and (2.12) yields

$$\tau'\, dx + c\rho A\, dx\, \ddot{w} + cp\ dx = \ddot{\phi}\rho\, dx\, I_p + \ddot{\phi} c^2 \rho A\, dx. \tag{2.13}$$

Now dividing by dx and rearranging, this becomes

$$\tau' + c\rho A\,(\ddot{w} - c\ddot{\phi}) + cp = \ddot{\phi}\rho I_p. \tag{2.14}$$

In Krenk and Høgsberg (2013), the following relation between torque and angle of twist is derived;

$$\tau = GK\frac{\partial \phi}{\partial x}, \tag{2.15}$$

where G is the shear modulus of the material, and K is the torsion constant (K will generally have to be computed numerically for these cross sections). This is an approximation to the more exact expression $\tau = GK\frac{\partial \phi}{\partial x} - R\frac{\partial^4 \phi}{\partial x^4}$, where the term $R\frac{\partial^4 \phi}{\partial x^4}$ comes from considering warping of the cross section. Here, warping of the cross section is ignored, and (2.15) is substituted into (2.14);

$$\frac{\partial}{\partial x}\left(GK\phi'\right) + c\rho A\,(\ddot{w} - c\ddot{\phi}) + cp = \ddot{\phi}\rho I_p. \tag{2.16}$$

Since we're considering a homogenous, uniform beam, the product GK is a constant. With this, and a bit of rearranging, we arrive at

$$GK\phi'' - \left(c^2\rho A + \rho I_p\right)\ddot{\phi} = -cp - c\rho A\ddot{w}, \tag{2.17}$$

which is the equation governing twist. Through a non-zero value of c, the twist is coupled with deflections due to bending, and to the vertical load p acting on the center of mass.

Together, the two coupled equations (2.8) and (2.17),

$$EI_z w'''' + \rho A \ddot{w} = -p + c\rho A \ddot{\phi}; \tag{2.18}$$

$$GK\phi'' - \left(c^2 \rho A + \rho I_p\right) \ddot{\phi} = -cp - c\rho A \ddot{w}, \tag{2.19}$$

comprises the set of governing dynamic equations of motion.

3
Solving the Coupled Equations of Motion

The coupled equations of motion derived in chapter 2 are solved using a series expansion. The aim is to be able to extract information about the coupling of the vibration, and visualize individual mode shapes of the vibration. A mode shape is the spatial shape of the beam associated with a single one of its natural frequencies. It is a certain linear combination of basis functions of the series expansion, oscillating at a single natural frequency.

The solution to the equations are represented by a series of basis functions, each multiplied by a time response function. This is done for deflections due to bending, as well as for angular deflection (twist) due to torsion. The basis functions could simply be chosen as sine or cosine functions, as in a fourier expansion. However, they are normally much better determined by paying attention to the boundary conditions. Since deriving spatial mode shapes by applying boundary conditions to the coupled equations of motion proves difficult, we shall instead derive them from the uncoupled equations of motion, by letting $c = 0$ in (2.18) and (2.19). These will then be used as a best guess for the basis functions of the series expansion.

After deciding on a set of basis functions, the differential equations are molded into a system of linear ordinary differential equations (ODEs), where the unknowns are the time response functions. When visualizing a natural response, the system of ODEs are cast as an an eigenvalue problem. The natural response is the beam responding to a set of initial conditions with no external force. When an external load is acting, called the forced response, the system is solved numerically by shaping it into a form suitable for use in MATLABs built-in functions for solving ODEs.

3.1 Deriving basis functions from the uncoupled equations

Letting $c = 0$ and $p = 0$ in (2.18) and (2.19), considering a natural response with no external force, leads to the uncoupled equations

$$EI_z w'''' + \rho A \ddot{w} = 0; \tag{3.1}$$

$$GK\phi'' - \rho I_p \ddot{\phi} = 0. \tag{3.2}$$

Basis functions for bending will be derived from (3.1) and basis functions for torsion will be derived from (3.2) in the following. Starting with bending, we attempt to write the solution to (3.1) in the form

$$w(x,t) = W(x)r(t), \tag{3.3}$$

where $w(x,t)$ is split up into a spatial part $W(x)$ and a temporal part $r(t)$. It will turn out that the complete solution $w(x,t)$ cannot be simply split up between a spatial part and a temporal part. The effort here will instead lead to many separate valid functions for $W(x)$. These will be called $W_n(x)$ as they depend on a parameter we will call n. $W_n(x)$ are the basis functions, and each of them will be multiplied by its corresponding temporal part $r_n(t)$. The complete solution $w(x,t)$ will then be a linear combination of these.

By substituting this into (3.1), we get

$$EI_z W''''(x)r(t) = -\rho A W(x)\ddot{r}(t). \tag{3.4}$$

Separation of variables leads to two equations

$$W''''(x)W(x)^{-1} = \alpha^4; \tag{3.5}$$

$$-\frac{\rho A}{EI_z}\ddot{r}(t)r(t)^{-1} = \alpha^4, \tag{3.6}$$

for a real constant α^4. Equation (3.5) is the one of interest here, as it will yield the basis functions. Avoiding the trivial solution $\alpha^4 = 0$, the solution is[1]

[1] Ole Christensen. *Differentialligninger og uendelige rækker*. 2005. ISBN 87-88-76473-7

$$W(x) = C_1 \sin(\alpha x) + C_2 \cos(\alpha x) + C_3 \sinh(\alpha x) + C_4 \cosh(\alpha x). \tag{3.7}$$

The value of α and the constants of integration C are found by applying boundary conditions to this equation. This process is shown in the following section for a simple set of boundary conditions.

3.1.1 *Basis functions for deflections due to bending of a hinged-hinged beam*

By a hinge we mean a support with the following boundary conditions:

$$\begin{aligned} w(0,t) &= 0; \\ w(L,t) &= 0; \\ w''(0,t) &= 0; \\ w''(L,t) &= 0. \end{aligned} \tag{3.8}$$

These boundary conditions for $w(x,t)$ translates directly into conditions for $W(x)$. To see this, take $w(0,t) = 0$ as an example;

$$w(0,t) = W(0)r(t) = 0. \tag{3.9}$$

Since the right hand side is zero, it must be true that either $W(0)$ or $r(t)$ be zero as well. Since $r(t) = 0$ is the trivial solution, which we are disregarding, $W(0)$ must be zero. Returning to equation (3.7), this means that

$$W(0) = C_2 + C_4 = 0; \tag{3.10}$$
$$W''(0) = -C_2\alpha^2 + C_4\alpha^2 = 0. \tag{3.11}$$

These equations imply $C_2 = C_4 = 0$. Turning to the remaining boundary conditions,

$$W(L) = C_1 \sin(\alpha L) + C_3 \sinh(\alpha L) = 0; \tag{3.12}$$
$$W''(L) = -C_1\alpha^2 \sin(\alpha L) + C_3\alpha^2 \sinh(\alpha L) = 0, \tag{3.13}$$

We see that eliminating α^2 from the second equation and adding the equations yields $2C_3 \sinh(\alpha l) = 0$, leading to $C_3 = 0$. (3.12) then claims

$$C_1 \sin(\alpha L) = 0. \tag{3.14}$$

This is the so-called frequency equation. Insisting that the last constant be non-zero, $C_1 \neq 0$, this means $\sin(\alpha L) = 0$, so the roots of the frequency equation are

$$\alpha L = n\pi, \tag{3.15}$$

or

$$\alpha = \frac{n\pi}{L}, \tag{3.16}$$

where n is an integer, but of course $n \neq 0$. α is the frequency of the basis function, as is evident from (3.7). The boundary conditions cannot assign a value to the remaining constant C_1, which represents the amplitude of the basis function. We shall leave this amplitude untouched for now and define it later. To summarize, we have

$$W_n(x) = D_n \sin\left(\frac{n\pi x}{L}\right), \tag{3.17}$$

where D_n is the amplitude which has yet to be defined. This defines a basis function for each n.

At this point, it is tempting to continue to solve (3.6) for $r(t)$. This we unfortunately cannot do, as we would have simply solved the uncoupled equations of motion. However, when substituted into equation (3.3), the result obtained above means that a specific solution to (3.3) $w_n(x,t)$ for a given n takes the form

$$w_n(x,t) = D_n \sin\left(\frac{n\pi x}{L}\right) r_n(t). \tag{3.18}$$

This represents an infinitude of solutions as n is unrestricted, so we shall have to express $w(x,t)$ as a linear combination of these;

$$w(x,t) = \sum_n D_n \sin\left(\frac{n\pi x}{L}\right) r_n(t). \tag{3.19}$$

As this is a linear combination, we should expect to find constants scaling one term in relation to the others. This is not the task of D_n.

We shall later wish to define D_n in a way which slightly simplifies the equations. Instead, the relative weight of the basis functions in the linear combination is taken care of by $r_n(t)$, which is as of yet undefined and shall scale accordingly when computed.

Note also from equation (3.17), that negative values of n really represent the same shapes as their positive counterpart. In other words, they are linearly dependent. As they are all summed in the linear combination of (3.19), it is unnecessary to consider negative values of n. When dropping the negative n, the adjustment of the amplitude of the remaining terms is accounted for by $r_n(t)$. Changing the index on the sum to only include postive integer values of n, we arrive at the final form for a hinged-hinged beam;

$$w(x,t) = \sum_{n=1}^{\infty} D_n \sin\left(\frac{n\pi x}{L}\right) r_n(t). \tag{3.20}$$

However, since (3.17) is valid only for a hinged-hinged beam, in the interest of being able to represent other boundary conditions we shall represent the deflection as

$$w(x,t) = \sum_{n=1}^{\infty} W_n(x) r_n(t). \tag{3.21}$$

If solving the uncoupled equations of motion was the goal, then $W_n(x)$ would be the mode shapes, as $r_n(t)$ would represent a harmonic oscillation at the n'th natural frequency. This is not the case for the coupled equations of motion. As we shall later see, $r_n(t)$ will be a mixture of harmonic oscillations at the various natural frequencies. $r_n(t)$ represents a mixture of natural frequencies instead of a single natural frequency, because the basis functions are derived from the uncoupled system.

In section 3.2, this series expansion will be substitued into the coupled equations of motion together with its counterpart for torsion, developed in the following section.

3.1.2 *Basis functions for a fixed-fixed beam with regards to torsion*

Running through the same procedure as for bending above, the simplest boundary conditions with respect to torsion is a fixed-fixed beam, allowing no twist in either end;

$$\begin{aligned} \phi(0,t) &= 0; \\ \phi(L,t) &= 0. \end{aligned} \tag{3.22}$$

Attempting to split up $\phi(x,t)$ into a spatial part $\Phi(x)$ and a temporal part $s(t)$;

$$\phi(x,t) = \Phi(x)s(t), \tag{3.23}$$

and inserting it into (3.2) yields

$$GK\Phi''(x)s(t) - \rho I_p \Phi(x)\ddot{s}(t) = 0. \tag{3.24}$$

Separation of variables leads to

$$\Phi''(x)\Phi(x)^{-1} = -\beta^2; \tag{3.25}$$

$$\frac{\rho I_p}{GK}\ddot{s}(t)s(t)^{-1} = -\beta^2, \tag{3.26}$$

for a real constant β^2. The harmonic solution of interest entails that $\beta^2 > 0$, whereby (3.25) has the solution

$$\Phi(x) = C_1 \cos(\beta x) + C_2 \sin(\beta x). \tag{3.27}$$

Applying the first boundary condition, $\phi(0,t) = 0$, we see that $C_1 = 0$. The second boundary condition $\phi(L,t) = 0$ implies $C_2 \sin(\beta L) = 0$, so to avoid the trivial solution, we must have

$$\sin(\beta L) = 0. \tag{3.28}$$

This is the frequency equation. The roots are

$$\beta L = n\pi, \tag{3.29}$$

or

$$\beta = \frac{n\pi}{L}. \tag{3.30}$$

The n'th basis function then take the form

$$\Phi_n(x) = F_n \sin\left(\frac{n\pi x}{L}\right), \tag{3.31}$$

where F_n is the amplitude. Wrapping this up in a linear combination as before, we arrive at

$$\phi(x,t) = \sum_{n=1}^{\infty} F_n \sin\left(\frac{n\pi x}{L}\right) s_n(t). \tag{3.32}$$

Expressing the basis functions as $\Phi_n(x)$ in order to account for other boundary conditions, (3.32) has the general form

$$\phi(x,t) = \sum_{n=1}^{\infty} \Phi_n(x) s_n(t). \tag{3.33}$$

Section 3.1.3 introduces some additional supports and boundary conditions.

3.1.3 *Other boundary conditions*

Bending support type	Boundary conditions
Hinged	$w = w'' = 0$
Clamped	$w = w' = 0$
Guided	$w' = w''' = 0$
Free	$w'' = w''' = 0$

Table 3.1: Considered support types for bending and their corresponding boundary conditions.

Torsion support type	Boundary conditions
Fixed	$\phi = 0$
Free	$\phi' = 0$

Table 3.2: Considered support types for torsion and their corresponding boundary conditions.

In addition to the simple boundary conditions of the previous two sections, We shall consider a number of additional support types. Table 3.1 contains a number of supports providing boundary conditions to bending. Table 3.2 contains the considered boundary conditions for torsion. Rather than go through the repetitious procedure of finding basis functions for the more advanced boundary conditions, they can be readily found in textbooks. Picking a few combinations of bending support types and borrowing the results[2] for the basis functions, the basis functions and roots are summed up in Tables 3.3 and 3.4.

[2] Jon Juel Thomsen. *Vibrations and Stability*. 2003. ISBN 978-3-642-07272-7

3.2 *Manipulating the coupled equations of motion from PDEs to ODEs*

With the basis functions established, we now go hunting for the time response functions $r_n(t)$ and $s_n(t)$. The series expansions for $w(x,t)$ and $\phi(x,t)$, equations (3.21) and (3.33), are inserted into the coupled equations of motion, equation (2.18) and (2.19);

$$EI_z \sum_{n=1}^{\infty} W_n''''(x) r_n(t) + \rho A \sum_{n=1}^{\infty} W_n(x) \ddot{r}_n(t) = -p + c\rho A \sum_{n=1}^{\infty} \Phi_n(x) \ddot{s}_n(t); \quad (3.34)$$

$$GK \sum_{n=1}^{\infty} \Phi_n''(x) s_n(t) - \left(c^2 \rho A + \rho I_p\right) \sum_{n=1}^{\infty} \Phi_n(x) \ddot{s}_n(t) = -cp - c\rho A \sum_{n=1}^{\infty} W_n(x) \ddot{r}_n(t). \quad (3.35)$$

Starting with (3.34), the equation is multiplied by one of the basis functions $W_k(x)$, and integrated over the length of the beam:

$$EI_z \sum_{n=1}^{\infty} \int_0^L W_k(x) W_n''''(x) \,\mathrm{d}x\, r_n(t) + \rho A \sum_{n=1}^{\infty} \int_0^L W_k(x) W_n(x) \,\mathrm{d}x\, \ddot{r}_n(t) = -\int_0^L W_k(x) p(x,t) \,\mathrm{d}x + c\rho A \sum_{n=1}^{\infty} \int_0^L W_k(x) \Phi_n(x) \,\mathrm{d}x\, \ddot{s}_n(t), \quad (3.36)$$

where the integration is performed over the individual terms in the summations. The integral featuring both $W_k(x)$ and $\Phi_n(x)$ will, along with a few constants, be denoted by $\psi_{k,n}$;

$$\psi_{k,n} = c\rho A \int_0^L W_k(x) \Phi_n(x) \,\mathrm{d}x, \quad (3.37)$$

Table 3.3: Bending basis functions for combinations of support types. The results are borrowed from Thomsen (2003).

Supports	Roots $\alpha_n L$ of frequency equation	Basis function without amplitude; $W_n(x)/D_n$
Hinged-hinged (simply supported)	$n\pi$	$\sin(\alpha_n x)$
Clamped-clamped (cantilever)	4.7300 7.8532 10.9956 14.1372 $\to (2n+1)\pi/2$	$J(\alpha_n x) - \frac{J(\alpha_n L)}{H(\alpha_n L)} H(\alpha_n x)$
Clamped-hinged	3.9266 7.0686 10.2102 13.3518 $\to (4n+1)\pi/4$	$J(\alpha_n x) - \frac{J(\alpha_n L)}{H(\alpha_n L)} H(\alpha_n x)$
Clamped-free	1.8751 4.6941 7.8548 10.9955 $\to (2n-1)\pi/2$	$J(\alpha_n x) - \frac{G(\alpha_n L)}{F(\alpha_n L)} H(\alpha_n x)$
Free-free	4.7300 7.8532 10.9956 14.1372 $\to (2n+1)\pi/2$	$G(\alpha_n x) - \frac{J(\alpha_n L)}{H(\alpha_n L)} F(\alpha_n x)$
Clamped-guided	2.3650 5.4978 8.6394 11.7810 $\to (4n-1)\pi/4$	$J(\alpha_n x) - \frac{F(\alpha_n L)}{J(\alpha_n L)} H(\alpha_n x)$

$J(u) = \cosh(u) - \cos(u)$;
$H(u) = \sinh(u) - \sin(u)$;
$G(u) = \cosh(u) + \cos(u)$;
$F(u) = \sinh(u) + \sin(u)$.

Table 3.4: Torsion basis functions for combinations of support types. The results are borrowed from Thomsen (2003).

Supports	Roots $\beta_n L$ of frequency equation	Basis function without amplitude; $\Phi_n(x)/F_n$
Fixed-fixed	$n\pi$	$\sin(\beta_n x)$
Fixed-free	$\frac{(2n-1)\pi}{2}$	$\sin(\beta_n x)$
Free-free	$n\pi$	$\cos(\beta_n x)$

so the first index on ψ is always the index on $W(x)$ and the second index is the index on $\Phi(x)$. Likewise, the integral featuring $W_k(x)$ and $p(x,t)$ will be denoted by $R_k(t)$:

$$R_k(t) = \int_0^L W_k(x)p(x,t)\,\mathrm{d}x. \tag{3.38}$$

Integration by parts is used on the first term on the left hand side, leading to

$$\begin{aligned} &-EI_z \sum_{n=1}^{\infty} \int_0^L W_k'(x)W_n'''(x)\,\mathrm{d}x + \left[W_k(x)W_n'''(x)\right]_0^L r_n(t) \\ &+\rho A \sum_{n=1}^{\infty} \int_0^L W_k(x)W_n(x)\,\mathrm{d}x\,\ddot{r}_n(t) \\ &\qquad = -R_k(t) + \sum_{n=1}^{\infty} \psi_{k,n}\ddot{s}_n(t). \end{aligned} \tag{3.39}$$

The term $[W_k(x)W_n'''(x)]_0^L$ is equal to zero because of the boundary conditions. That this term is equal to zero is obvious when looking at the boundary conditions in Table 3.1. Performing integration by parts once more leads to

$$\begin{aligned} EI_z \sum_{n=1}^{\infty} \int_0^L W_k''(x)W_n''(x)\,\mathrm{d}x\,r_n(t) + \rho A \sum_{n=1}^{\infty} \int_0^L W_k(x)W_n(x)\,\mathrm{d}x\,\ddot{r}_n(t) \\ = -R_k(t) + \sum_{n=1}^{\infty} \psi_{k,n}\ddot{s}_n(t). \end{aligned} \tag{3.40}$$

Again the term $\left[W_k'(x)W_n''(x)\right]_0^L$ is zero. The integrals can be simplified radically by utilizing orthogonality conditions for the differential equation (3.5) from which the basis functions were derived. These conditions tells us that the integrals only hold non-zero values when $k = n$, that is,

$$\int_0^L W_k W_n\,\mathrm{d}x = 0, \quad \text{for } k \neq n, \tag{3.41}$$

and

$$\int_0^L W_k'' W_n''\,\mathrm{d}x = 0, \quad \text{for } k \neq n. \tag{3.42}$$

These conditions of orthogonality are reviewed in more detail in the appendix, section A.2. When $k = n$, we shall like the first integral to be equal to one, in order to simplify the equations;

$$\int_0^L W_k W_n\,\mathrm{d}x = 1, \quad \text{for } k = n. \tag{3.43}$$

This implies that

$$\int_0^L W_k'' W_n''\,\mathrm{d}x = \alpha_n^4, \quad \text{for } k = n, \tag{3.44}$$

see (A.6). We have the freedom to dictate that the first integral be equal to one, because we can assign values to the amplitudes D_n of the basis functions in equation (3.17), such that the integral is equal to one. That is, we define

$$D_n = \frac{1}{\sqrt{\int_0^L W_n(x)^2 \, \mathrm{d}x}}, \tag{3.45}$$

where $W_n(x)$ in this equation represents the basis function without its amplitude, as this would otherwise be a recursive equation. In other words, here $W_n(x)$ is the formula shown in Table 3.3. With the introduction of the kronecker delta,

$$\delta_{kn} = \begin{cases} 0 & \text{for } k \neq n; \\ 1 & \text{for } k = n, \end{cases} \tag{3.46}$$

this can summed up as

$$\int_0^L W_k(x) W_n(x) \, \mathrm{d}x = \delta_{kn}, \tag{3.47}$$

and

$$\int_0^L W_k''(x) W_n''(x) \, \mathrm{d}x = \delta_{kn} \alpha_n^4. \tag{3.48}$$

Using (3.47) and (3.48) in (3.40) results in

$$EI_z \sum_{n=1}^{\infty} \delta_{kn} \alpha_n^4 r_n(t) + \rho A \sum_{n=1}^{\infty} \delta_{kn} \ddot{r}_n(t) = -R_k(t) + \sum_{n=1}^{\infty} \psi_{k,n} \ddot{s}_n(t). \tag{3.49}$$

The kronecker delta eliminates all but one term in the summations;

$$EI_z \alpha_k^4 r_k(t) + \rho A \ddot{r}_k(t) = -R_k(t) + \sum_{n=1}^{\infty} \psi_{k,n} \ddot{s}_n(t). \tag{3.50}$$

This equation couples any specific bending basis function $W_k(x)$ and its associated time response function $r_k(t)$ to the torsion basis functions through the second term on the right hand side. The equation is valid for all positive integer values of k; each corresponding to the k'th basis function.

As we cannot include an infinite number of terms in the series expansion of $w(x,t)$, we shall have to include only some finite number of terms, N. Smaller values of k correspond to lower spatial frequencies in the basis functions, so values of k going from 1 to N will represent the N basis functions with the lowest spatial frequencies. Note also that importantly, this equation now only depends on time t, as the integrals eliminate any dependence on x. This is now an ordinary differential equation, not a partial one. At the end of this section we shall construct a system of ODEs written as a matrix equation, but first the equations of torsion must be brought up to speed.

The coupled equation of motion for torsion (3.35) is tackled in much the same manner as equation (3.34) was in the above. The following is dealt with rather cursory as it is mostly repeating the same procedure.

Equation (3.35) is multiplied by a basis function $\Phi_k(x)$, integrated over the length of the beam and rewritten by performing integration by parts to produce

$$\begin{aligned} &- GK \sum_{n=1}^{\infty} \int_0^L \Phi_k'(x)\Phi_n'(x)\,\mathrm{d}x\, s_n(t) \\ &- \left(c^2\rho A + \rho I_p\right) \sum_{n=1}^{\infty} \int_0^L \Phi_k(x)\Phi_n(x)\,\mathrm{d}x\, \ddot{s}_n(t) \\ &\quad = -c \int_0^L \Phi_k(x)p(x,t)\,\mathrm{d}x - c\rho A \sum_{n=1}^{\infty} \int_0^L \Phi_k(x)W_n(x)\,\mathrm{d}x\, \ddot{r}_n(t). \end{aligned} \tag{3.51}$$

The integral featuring $\Phi_k(x)$ and $W_n(x)$ along with its scalar factors can be expressed as $\psi_{n,k}$ and the integral featuring $\Phi_k(x)$ and $p(x,t)$ along with its scalar factor c will be denoted by $S_k(t)$;

$$S_k(t) = c \int_0^L \Phi_k(x)p(x,t)\,\mathrm{d}x. \tag{3.52}$$

The following orthogonality conditions are valid for the torsion basis functions, and are reviewed in section A.2;

$$\int_0^L \Phi_k\Phi_n\,\mathrm{d}x = 0, \quad \text{for } k \neq n, \tag{3.53}$$

and

$$\int_0^L \Phi_k'\Phi_n'\,\mathrm{d}x = 0, \quad \text{for } k \neq n. \tag{3.54}$$

The amplitudes F_n of the torsion basis functions of equation (3.31), are defined such that

$$\int_0^L \Phi_k(x)\Phi_n(x)\,\mathrm{d}x = \delta_{kn}. \tag{3.55}$$

Which means that the constants are computed by

$$F_n = \frac{1}{\sqrt{\int_0^L \Phi_n(x)^2\,\mathrm{d}x}}, \tag{3.56}$$

where $\Phi_n(x)$ in this equation represents the basis function without its amplitude, as this would otherwise be a recursive equation. In other words, here $\Phi_n(x)$ is the formula shown in Table 3.4. Additionally,

$$\int_0^L \Phi_k'(x)\Phi_n'(x)\,\mathrm{d}x = \delta_{kn}\beta_n^2, \tag{3.57}$$

which can be seen from equation (A.16). Substituting the above into equation (3.51) leads to

$$\begin{aligned} &- GK \sum_{n=1}^{\infty} \delta_{kn}\beta_n^2 s_n(t) - \left(c^2\rho A + \rho I_p\right) \sum_{n=1}^{\infty} \delta_{kn}\ddot{s}_n(t) \\ &\qquad = -S_k(t) - \sum_{n=1}^{\infty} \psi_{n,k}\ddot{r}_n(t). \end{aligned} \tag{3.58}$$

Note that the indices on $\psi_{n,k}$ have been switched compared to equation (3.50). Letting the kronecker delta eliminate the summations followed by reversing the signs yields

$$GK\beta_k^2 s_k(t) + \left(c^2\rho A + \rho I_p\right)\ddot{s}_k(t) = S_k(t) + \sum_{n=1}^{\infty} \psi_{n,k}\ddot{r}_n(t), \tag{3.59}$$

This equation, in combination with (3.50), provides one equation for every unknown time response function, allowing us to combine the coupled bending and torsion equations into a system of linear ODEs.

Taking only the first N terms in the series expansion, the linear ordinary differential equations represented by (3.50) and (3.59) are written as a matrix equation. All the factors of the terms $\ddot{r}_k(t)$ or $\ddot{s}_k(t)$ are collected into a mass matrix $\mathbf{M}$. All factors of $r_k(t)$ or $s_k(t)$ are collected into a stiffness matrix $\mathbf{K}$, and the terms $R_k(t)$ and $S_k(t)$ that comes from the external load are collected into a vector $\mathbf{f}$ on the right hand side. The matrix equation looks as follows:

$$\mathbf{M}\ddot{\mathbf{z}} + \mathbf{K}\mathbf{z} = \mathbf{f}, \tag{3.60}$$

where $\mathbf{z}$ is the vector of time response functions;

$$\mathbf{z} = \begin{bmatrix} r_1 & \dots & r_N & s_1 & \dots & s_N \end{bmatrix}^\top, \tag{3.61}$$

where $\top$ indicates a transpose. The mass matrix is

$$\mathbf{M} = \begin{bmatrix} \operatorname{diag}(\rho A) & \mathbf{\Psi} \\ \mathbf{\Psi}^\top & \operatorname{diag}\left(c^2\rho A + \rho I_p\right) \end{bmatrix}, \tag{3.62}$$

where diag$(\dots)$ is a diagonal matrix. Here the diagonnal matrices are N by N matrices. $\mathbf{\Psi}$ is the matrix

$$\mathbf{\Psi} = \begin{bmatrix} -\psi_{1,1} & -\psi_{1,2} & \cdots & -\psi_{1,N} \\ -\psi_{2,1} & -\psi_{2,2} & \cdots & -\psi_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ -\psi_{N,1} & -\psi_{N,2} & \cdots & -\psi_{N,N} \end{bmatrix}. \tag{3.63}$$

The stiffness matrix $\mathbf{K}$ is

$$\mathbf{K} = \operatorname{diag}\left(EI_z\alpha_1^4, \dots, EI_z\alpha_N^4, GK\beta_1^2, \dots, GK\beta_N^2\right). \tag{3.64}$$

The right hand side $\mathbf{f}$ is

$$\mathbf{f} = \begin{bmatrix} -R_1(t) & \dots & -R_N(t) & S_1(t) & \dots & S_N(t) \end{bmatrix}^\top. \tag{3.65}$$

In this section, the PDEs have been converted into a system of ordinary differential equations. This system is solved in the following sections in different ways for different purposes. When we want to visualize an individual mode shape or a natural response, the matrix equation is cast as an eigenvalue problem. The natural response is the problem of finding a response to a set of initial conditions when no external force is acting. When there is an external force acting, the response of the beam is called the forced response. The vibration is computed numerically, by converting the matrix equation (3.60) into a system of first order ODEs that can be solved numerically in MATLAB.

3.3 *Harmonic oscillations*

In this section, the matrix equation is cast as an eigenvalue problem. First, the external load is set to zero $p = 0$, after which the matrix equation (3.60) takes the form

$$\mathbf{M}\ddot{\mathbf{z}} + \mathbf{K}\mathbf{z} = \mathbf{0}. \tag{3.66}$$

This is cast as a generalized eigenvalue problem by first substituting with $\mathbf{z} = e^{i\omega t}\mathbf{v}$, where i is the imaginary unit. Substituting and rearranging leads to the eigenvalue problem

$$\mathbf{K}\mathbf{v} = \omega^2\mathbf{M}\mathbf{v}. \tag{3.67}$$

ω is a natural frequency and ω^2 is an eigenvealue. As $\mathbf{M}$ and $\mathbf{K}$ are of size $2N$ by $2N$, this produces $2N$ sets of natural frequencies and eigenvectors. The eigenvectors $\mathbf{v}$ and eigenvalues ω^2 are computed in MATLAB. Since each eigenvalue corresponds to two solutions, $\mathbf{z} = e^{i\omega t}\mathbf{v}$ and $\mathbf{z} = e^{-i\omega t}\mathbf{v}$, the solutions can be written as

$$\mathbf{z}_k = \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right)\mathbf{v}_k, \tag{3.68}$$

where k means that this is the k'th eigenvector $\mathbf{v}_k$ and natural frequency ω_k. A more detailed justification for writing the solution in the form (3.68) is provided in Inman[3]. Here, A_k and B_k are constants to be determined by the initial conditions. Since k represents any value from 1 to $2N$, A_k and B_k add up to $4N$ constants.

[3] Daniel. Inman. *Engineering Vibration*. Pearson Prentice Hall, 2008. ISBN 0132281732, 9780132281737

Note that $\mathbf{z}_k$ of equation (3.68) has a significant importance. $\mathbf{z}_k$ is the temporal function of the k'th mode shape, as $\mathbf{z}_k$ is a vector of time response functions $r_k(t)$ and $s_k(t)$ which oscillate at the k'th natural frequency ω_k. To summarize, $\mathbf{z}_k$ describes the vibration of the k'th mode shape over time. The eigenvector $\mathbf{v}_k$ determines the relative weight of the time response functions $r_k(t)$ and $s_k(t)$ at the natural frequency ω_k. Recall from equations (3.21) and (3.33) that this means the eigenvector $\mathbf{v}_k$ indirectly controls the relative weight of the basis functions $W_n(x)$ and $\Phi_n(x)$ for the k'th mode shape.

It is not yet immediately obvious how the k'th mode shape looks like, (isolating a single mode shape is the purpose of section 3.3.1).

Equation (3.68) represents $2N$ different solutions to (3.66), since k represents any value from 1 to $2N$. By linear combination the complete solution for the time responses $\mathbf{z}$ is

$$\mathbf{z} = \sum_{k=1}^{2N} \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right)\mathbf{v}_k. \tag{3.69}$$

This is a vector equation, but breaking up the eigenvector we see from (3.61) that solutions for specific time response functions $r_n(t)$ or $s_n(t)$ are

$$r_n(t) = \sum_{k=1}^{2N} \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right) v_{n,k}; \tag{3.70}$$

$$s_n(t) = \sum_{k=1}^{2N} \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right) v_{N+n,k}. \tag{3.71}$$

where $v_{n,k}$ is the n'th element of the eigenvector $\mathbf{v}_k$ corresponding to the k'th natural frequency ω_k. When two indices are used as in $v_{n,k}$, the second will denote the eigenvector, while the first will denote the element in that eigenvector.

The full solution for bending is then

$$\begin{aligned} w(x,t) &= \sum_{n=1}^{N} W_n(x) r_n(t) \\ &= \sum_{n=1}^{N} W_n(x) \sum_{k=1}^{2N} \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right) v_{n,k}. \end{aligned} \tag{3.72}$$

Similarly, for torsion it is

$$\begin{aligned} \phi(x,t) &= \sum_{n=1}^{N} \Phi_n(x) s_n(t) \\ &= \sum_{n=1}^{N} \Phi_n(x) \sum_{k=1}^{2N} \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right) v_{N+n,k}. \end{aligned} \tag{3.73}$$

In other words, if we can determine the constants A_k and B_k, the two equations above fully describes the vibration.

The following two sections concern determination of the constants A_k and B_k. Section (3.3.1) will determine the constants from the perspective of wanting to only excite a specific mode shape. Section (3.3.2) determines the constants from the perspective of starting the vibration from a set of initial conditions.

3.3.1 *Mode shapes*

A mode shape is the spatial shape of the beam as it performs harmonic oscillations at a single natural frequency. The real motion of the beam is a linear combination of the mode shapes. A single mode shape will combine spatial basis functions from both the series expansion for bending and the series expansion for torsion, since the beam features coupled bending and torsion vibrations. In other words, a mode shape is a linear combination of all $W_n(x)$ and $\Phi_n(x)$, oscillating at a single natural frequency.

In the interest of visualizing independent mode shapes, eliminate all but one mode shape from (3.69) by letting all the constants be equal to zero except the two constants A_k and B_k corresponding to the k'th mode shape. Equations (3.72) and (3.73) subsequently become

$$w(x,t) = \sum_{n=1}^{N} W_n(x) \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right) v_{n,k}; \tag{3.74}$$

$$\phi(x,t) = \sum_{n=1}^{N} \Phi_n(x) \left(A_k \sin(\omega_k t) + B_k \cos(\omega_k t)\right) v_{N+n,k}, \tag{3.75}$$

The constants of integration A_k and B_k determine amplitude and phase of the oscillation, and we shall choose $A_k = 0$ in the interest of starting the oscillation at its greatest amplitude for time $t = 0$.

With A_k eliminated, the solution takes the form

$$w_k(x,t) = B_k \cos(\omega_k t) \sum_{n=1}^{N} W_n(x) v_{n,k}; \tag{3.76}$$

$$\phi_k(x,t) = B_k \cos(\omega_k t) \sum_{n=1}^{N} \Phi_n(x) v_{N+n,k}. \tag{3.77}$$

B_k will be chosen so that the maximum rotation of the beam is small enough that it does not violate the assumption of small deflections which keeps the problem linear.

3.3.2 *Natural response*

The natural response is the time response of a beam subjected to initial conditions with no external load. The natural response will combine several mode shapes. It is calculated by determining suitable values for the constants of integration A_k and B_k. Returning to equation (3.72) and (3.73), the constants can be determined by applying initial conditions, $w(x,0)$, $\dot{w}(x,0)$, $\phi(x,0)$ and $\dot{\phi}(x,0)$. In the case of bending, recall from (3.21) that the first initial condition $w(x,0)$ can be written as

$$w(x,0) = \sum_{n=1}^{N} W_n(x) r_n(0). \tag{3.78}$$

Multiply this equation by a basis function $W_j(x)$, and integrate from 0 to L.

$$\begin{aligned}\int_0^L W_j(x) w(x,0)\, dx &= \int_0^L W_j(x) \sum_{n=1}^{N} W_n(x) r_n(0)\, dx \\ &= \sum_{n=1}^{N} \int_0^L W_j(x) W_n(x)\, dx\, r_n(0)\end{aligned} \tag{3.79}$$

Recall that the amplitudes of the basis functions $W_j(x)$ have been chosen so that the integral on the right hand side is equal to unity. The previously mentioned orthogonality conditions (equation (3.41)) also applies, and used together these conditions reduces the right hand side:

$$\int_0^L W_j(x) w(x,0)\, dx = r_j(0). \tag{3.80}$$

Now substitute the expression for $r_j(0)$ found in equation (3.70) into the above:

$$\begin{aligned}\int_0^L W_j(x) w(x,0)\, dx &= \sum_{k=1}^{2N} \left(A_k \sin(\omega_k 0) + B_k \cos(\omega_k 0)\right) v_{j,k} \\ &= \sum_{k=1}^{2N} B_k v_{j,k}.\end{aligned} \tag{3.81}$$

This represents N equations, since j can take any value from one to N. (Even though there are values of $v_{j,k}$ for larger j, $W_j(x)$ on the left hand side of (3.81) only allows j to range from 1 to N, and the substitution by equation (3.70) would not make sense for larger j either. The rest of the elements $v_{j,l}$ of the eigenvectors are used with the next initial condition).

Using equation (3.71) and the initial condition $\phi(x,0)$ in the exact same way, we obtain

$$\int_0^L \Phi_j(x)\phi(x,0)\ \mathrm{d}x = \sum_{k=1}^{2N} B_k v_{N+j,k}. \tag{3.82}$$

This is an additional N equations, amounting to a system of $2N$ linear equations with the $2N$ unknown constants $B_1, \ldots, B_{2N}$. The constants B_k can be determined from the above alone, but let us first find expressions for A_k before doing so.

Repeating the procedure starting with $\dot{w}(x,0)$, yields

$$\begin{aligned}\int_0^L W_j(x)\dot{w}(x,0)\ \mathrm{d}x &= \int_0^L W_j(x) \sum_{n=1}^{N} W_n(x)\dot{r}_n(0)\ \mathrm{d}x \\ &= \dot{r}_j(0).\end{aligned} \tag{3.83}$$

Differentiating (3.70) with respect to time and substituting gives

$$\begin{aligned}\int_0^L W_j(x)\dot{w}(x,0)\ \mathrm{d}x &= \sum_{k=1}^{2N} \left(A_k\omega_k \cos(\omega_k 0) - B_k\omega_k \sin(\omega_k 0)\right) v_{j,k} \\ &= \sum_{k=1}^{2N} A_k\omega_k v_{j,k}.\end{aligned} \tag{3.84}$$

This is followed up by similar equations for the initial condition $\dot{\phi}(x,0)$:

$$\int_0^L \Phi_j(x)\dot{\phi}(x,0)\ \mathrm{d}x = \sum_{k=1}^{2N} A_k\omega_k v_{N+j,k}. \tag{3.85}$$

In summation, equations (3.81)–(3.85) determine the constants of integration, A_k and B_k, through two linear systems of $2N$ equations each. The system that determines the constants B_k is

$$\begin{bmatrix} v_{1,1} & \cdots & v_{1,2N} \\ \vdots & \ddots & \vdots \\ v_{2N,1} & \cdots & v_{2N,2N} \end{bmatrix} \begin{bmatrix} B_1 \\ \vdots \\ B_{2N} \end{bmatrix} = \begin{bmatrix} \int_0^L W_1(x)w(x,0)\ \mathrm{d}x \\ \vdots \\ \int_0^L W_N(x)w(x,0)\ \mathrm{d}x \\ \int_0^L \Phi_1(x)\phi(x,0)\ \mathrm{d}x \\ \vdots \\ \int_0^L \Phi_N(x)\phi(x,0)\ \mathrm{d}x \end{bmatrix}. \tag{3.86}$$

The system that determines the constants A_k is

$$\begin{bmatrix} \omega_1 v_{1,1} & \cdots & \omega_{2N} v_{1,2N} \\ \vdots & \ddots & \vdots \\ \omega_1 v_{2N,1} & \cdots & \omega_{2N} v_{2N,2N} \end{bmatrix} \begin{bmatrix} A_1 \\ \vdots \\ A_{2N} \end{bmatrix} = \begin{bmatrix} \int_0^L W_1(x)\dot{w}(x,0)\ \mathrm{d}x \\ \vdots \\ \int_0^L W_N(x)\dot{w}(x,0)\ \mathrm{d}x \\ \int_0^L \Phi_1(x)\dot{\phi}(x,0)\ \mathrm{d}x \\ \vdots \\ \int_0^L \Phi_N(x)\dot{\phi}(x,0)\ \mathrm{d}x \end{bmatrix}. \tag{3.87}$$

Finding the solution to this system can easily be achieved by computing the inverse of the matrix, or by using built in tools like the backslash \ operator in MATLAB. Once the solution to this system has been computed, $w(x,t)$ and $\phi(x,t)$ are computed from equations (3.72) and (3.73).

3.4 *Forced response*

The forced response is the time response of a beam subjected to an external load and started from rest. When an external load is applied $p(x,t)$, the matrix equation (3.60) is inhomogenous. Since the beam is initially at rest for $t = 0$, the vibration comes solely from the external force. The solution is simply the particular solution to the inhomogenous system. Unfortunately, the particular solution is difficult to find in the general case, since the user of the software may specify any function of x and t. For this reason, the solution is obtained by numerical means.

The following manipulates the system into a form that is readily solved by MATLAB's functions for numerically solving systems of ordinary differential equations. MATLAB requires that the system is in the form of first order ODEs.

Returning to equation (3.60), let

$$\mathbf{y} = \dot{\mathbf{z}}. \tag{3.88}$$

Substituting with this gives

$$\mathbf{M}\dot{\mathbf{y}} + \mathbf{K}\mathbf{z} = \mathbf{f}. \tag{3.89}$$

Together, equations (3.88) and a rearrangement of (3.89) provide a system of first order linear ODEs:

$$\dot{\mathbf{z}} = \mathbf{y}; \tag{3.90}$$

$$\dot{\mathbf{y}} = -\mathbf{M}^{-1}\mathbf{K}\mathbf{z} + \mathbf{M}^{-1}\mathbf{f}. \tag{3.91}$$

Which when written in matrix form is

$$\begin{bmatrix} \dot{\mathbf{z}} \\ \dot{\mathbf{y}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\mathbf{M}^{-1}\mathbf{K} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \mathbf{y} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{M}^{-1}\mathbf{f} \end{bmatrix}, \tag{3.92}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{0}$ is either a vector or a matrix of zeros, depending on the context. The vector $\left[\begin{bmatrix} \mathbf{z} & \mathbf{y} \end{bmatrix}^\top\right]$ has $4N$ elements, and the coefficient matrix is $4N$ by $4N$ as well. The goal here is still to find $\mathbf{z}$.

This can be further condensed by letting

$$\mathbf{q} = \begin{bmatrix} \mathbf{z} \\ \mathbf{y} \end{bmatrix}, \tag{3.93}$$

$$\mathbf{H} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\mathbf{M}^{-1}\mathbf{K} & \mathbf{0} \end{bmatrix}, \tag{3.94}$$

and

$$\mathbf{b} = \begin{bmatrix} \mathbf{0} \\ \mathbf{M}^{-1}\mathbf{f} \end{bmatrix}. \tag{3.95}$$

Resulting in the short form

$$\dot{\mathbf{q}} = \mathbf{H}\mathbf{q} + \mathbf{b}. \tag{3.96}$$

This form is well suited for implementation in MATLAB. Once $\mathbf{z}$ has been found numerically and the solutions for $r_k(t)$ and $s_k(t)$ have been extracted from $\mathbf{z}$, the solutions for $w(x,t)$ and $\phi(x,t)$ are again given by (3.21) and (3.33) as

$$w(x,t) = \sum_{n=1}^{N} W_n(x) r_n(t); \tag{3.97}$$

$$\phi(x,t) = \sum_{n=1}^{N} \Phi_n(x) s_n(t). \tag{3.98}$$

Section 3.3 and the present section described solutions to three problems. The problem of visualizing a specific mode shape, computing the natural response and computing the forced response. The implementation of the solutions is the topic of chapter 4.

4 Implementation in MATLAB

The present chapter will implement the solutions found in the previous chapter as well as outline some of the more interesting design considerations and a few of the struggles that went into writing the code. The implementation of the solutions are discussed first, followed by the design of the GUI discussed in section 4.4 (a sneak peak of the GUI can be had from Figure 4.1). Section 4.1 contains a detailed discussion of the computation of the solution by use of the results from chapter 3. The animations deserve a section as well. The way in which the animations are produced is described in section 4.6.

Not every part of the code can be described in this chapter, and some of it is fairly trivial anyway. The complete matlab code is provided in the appendix, chapter B. The program was written and tested in MATLAB version R2013b.

The code is available as a zip file.[1]

[1] The program code. URL `https://dl.dropboxusercontent.com/u/7180193/BA/BAprogram.zip`

4.1 Computing the vibration

In the program code, the following computation of the vibration all takes place in the file `solver.m`.

A one dimensional array `xpoints` is defined. It contains a bunch of x-coordinates from 0 to L. It is defined with a default resolution of approximately 500 points. It should provide about one point per pixel along the x axis in the final animation. Likewise, `tpoints` is defined as a one dimensional array holding time values for each frame.

The computation of the basis functions $W_n(x)$ and $\Phi_n(x)$ are ordered into functions with a function for each combination of support types. As an example, pseudo code for a function which computes bending basis functions for a clamped-clamped beam looks like

```
function [modeshape, spatialfreq] = clamped_clamped_BENDING()
% 1. Define variable "roots" that holds roots to the frequency
    equation.
        precomputed = [4.7300 7.8532 10.9956 14.1372];
        if N<5
            roots = precomputed(1:N);
```

```
        else
            roots = [precomputed (2*(5:N)+1)*pi/2];
        end
% 2. Use MATLAB function ndgrid() to convert variables "roots" and
     "xpoints" into arrays.
        [rG, xG] = ndgrid(roots,xpoints);
% 3. Compute basis functions without amplitude factors.
        basisfunctionRaw = cosh(rG.*xG/L)-cos(rG.*xG/L)-(cosh(rG)-
            cos(rG))./(sinh(rG)-sin(rG)).*(sinh(rG.*xG/L)-sin(rG
            .*xG/L));
% 4. Compute amplitude factors D_n.
        NormalizationFactor = 1./sqrt(trapz(xpoints,(
            basisfunctionRaw.^2)'));
% 5. Multiply the amplitude factors onto the basis functions.
        basisfunction = diag(NormalizationFactor)*basisfunctionRaw
            ;
% 6. Return basis functions as a two dimensional array and return
    spatial frequencies as a one dimensional array.
        spatialfreq = roots/L;
end
```

The roots defined in step 1. and the basis functions without amplitudes D_n defined in step 3. come from Table 3.3. Since there are different basis functions $W_n(x)$ as indicated by the index n, and since they depend on x, we will represent the basis functions returned by this function as an array. The array will be structured so that the first row is $W_1(x)$, the second row is $W_2(x)$ and so on. The columns will correspond to the x-coordinates of the variable `xpoints`. In order structure the array in this way, the vectors `roots` and `xpoints` are first converted into arrays that can then be multiplied or divided like scalars. This is achieved using MATLAB's `ndgrid` command in step 2.

Step 4. computes the amplitude factors as described in equation (3.45). The basis functions are corrected with these amplitudes in step 5. The roots are also returned along with the basis functions in step 6., as they define α_n.

`ndgrid` is used repeatedly in the rest of the code to represent functions of two variables as a numerical array, or to represent several functions of a single variable with an index in the same array. The decision to represent functions as numerical arrays instead of as continous functions is discussed in section 4.7.

Now, continuing on, the matrix Ψ, which occurs in the mass matrix $\mathbf{M}$ of equation (3.60), is computed. Recall from equation (3.37) that the elements of Ψ are integrals which couple the bending basis functions with the torsion basis functions. The elements of Ψ are computed by numerical integration using the MATLAB command `trapz`.

```
for i=1:N
    for j=1:N
        Psi(i,j) = c*rho*area*trapz(xpoints,W(i,:).*Phi(j,:));
    end
end
```

Then the mass matrix M and stiffness matrix K are assembled from equations (3.62) and (3.64).

The next step is to solve the eigenvalue problem. The eigenvalue problem is solved by use of MATLAB's `eig`. This returns the natural frequencies and eigenvectors:

```
function [natfreq,eigenvectors] = EigenProblemSolver(M,K)
% 1. Solve eigenvalue problem with eig():
        [eigenvectors,eigenvalues] = eig(K,M);
% 2. A vector "natfreq" is created which are the square roots of
     the eigenvalues.
% 3. The function returns the natural frequencies and the
     eigenvectors.
end
```

At this point in the `solver.m` main function, one of three functions are called depending on whether the aim is to visualize a single mode shape, the natural response or the forced response. The following three sections describe these cases.

4.1.1 *Computing a single mode shape*

When visualizing a single mode shape, a call is made to a function in `solver.m` called `singlemodeshape`.

In implementing equations (3.76) and (3.77), the sum is computed first. The sum depends only on x for a given mode shape k, so it can be represented as a one-dimensional array with each value representing the value of the sum for a certain value of x. First, taking (3.76) as an example, without actually performing the summation yet, the *content* of the sum is computed into an array `sumcontent` where rows represent values of n and columns represent x-coordinates:

```
for i = 1:N
    sumcontent(i,:) = W(i,:) * eigenvectors(i,data.modeshape);
end
```

`data.modeshape` is any number from 1 to $2N$. It is the mode shape that the user has currently chosen to visualize. Next, the actual sum is the one-dimensional array computed by the MATLAB command `sum` and stored in the variable `sumterm`:

```
sumterm = sum(sumcontent);
```

Note that the above lines of code is an outline of the full code, since some special cases have to be taken into account in several places. This is true in general in this chapter in order to keep it brief by focusing on the important lines of code. The full code can always be seen in the appendix B. The excerpts highlighted here only attempts to illustrate the general idea behind the code.

Now, working through the rest of equation (3.76), the constant B_k will be chosen so that in the visualization of the mode shapes,

$$\max(\phi_k(x,t)) = 0.1. \tag{4.1}$$

It is however much easier to ignore B_k for now, and once the complete animation is computed, the whole solution is scaled to respect this condition. Worrying only about the cosine term of (3.76), the `ndgrid` function again becomes useful as the cosine term depends on the temporal variable t while the summation term depends on the spatial variable x.

```
[sumtermGRID,tpointsGRID] = ndgrid(sumterm,tpoints);
w = cos(natfreq(data.modeshape)*tpointsGRID) .* sumtermGRID;
```

`w` now is a two-dimensional array representing the solution $w(x,t)$. The rows of `w` represent x-coordinates and the columns represent time values or frames in the animation.

Finally the solution is scaled to comply with (4.1). Computing $\phi(x,t)$ is done completely analogous to $w(x,t)$.

4.1.2 *Natural response*

If the natural response is chosen, the vibration is computed by the function `naturalresponse`. A general outline of this function is given below.

The initial conditions are given as functions of x by the user. These are turned into one-dimensional arrays. For example the initial condition $w(x,0)$, which is inputted by the user as a continuous function, is converted into a one-dimensional array of points by computing the function value for every x-coordinate in `xpoints`:

```
initialw = data.initialw(xpoints);
```

where `data.initialw` is the function for $w(x,0)$ specified by the user. Next an array `Ww` is computed, which is an array representing the product $W_k(x)w(x,0)$, occuring on the right hand side of equation (3.86). The rows in the array represent values of k, while the columns represent x-coordinates.

```
for i = 1:N
    Ww(i,:) = W(i,:) .* initialw;
end
```

Equivalently arrays `Phiphi`, `Wwdot` and `Phiphidot` are computed to match the other integrals of equations (3.86) and (3.87). The complete right hand side of (3.86) is computed and the system is solved for B by the MATLAB operator \.

```
rhs = [trapz(xpoints,Ww') trapz(xpoints,Phiphi')]';
B = eigenvectors\rhs;
```

Note that the coefficient matrix of (3.86) is simply the array of eigenvectors returned by `eig`. Computation of the constants A_k of (3.87) is very similar in nature. It is done by

```
rhs = [trapz(xpoints,Wwdot') trapz(xpoints,Phiphidot')]';
A = (eigenvectors*diag(natfreq(:)))\rhs;
```

What remains is to use equation (3.72) and (3.73) to compute $w(x,t)$ and $\phi(x,t)$. For w, this is done by two loops, looping over the outer summation and the inner summation, as well as again utilizing `ndgrid` to represent the function of two variables $w(x,t)$ as an array:

```
for n = 1:N
    sumterm = 0;
    for k = 1:2*N
        sumterm = sumterm + (A(k)*sin(natfreq(k)*tpoints)+B(k)*cos
            (natfreq(k)*tpoints)) * eigenvectors(n,k);
    end
    [WGRID,sumtermGRID] = ndgrid(W(n,:),sumterm);
    w = w + WGRID .* sumtermGRID;
end
```

where `sumterm` being computed by the inner loop, is the inner summation of (3.72) depending only on t.

4.1.3 *Forced response*

The forced response is computed by the function `forcedresponse` also found in `solver.m`. First $\mathbf{H}$ of equation (3.94) is assembled, which is straightforward:

```
H = [zeros(2*N) eye(2*N); -M\K zeros(2*N)];
```

The right hand side of equation (3.96) is computed by the nested function `odeRHS`, which is used in MATLAB's ODE solver `ode45`. The external load specified by the user as a continuous function, is converted into a one-dimensional array, the same as the initial conditions for the natural response in the previous section.

The function `odeRHS` that computes the right hand side of (3.96) has the structure

```
function odeRHS = RHS(t,q)
% 1. Convert external load function to one-dimensional array:
        pvec = data.p(xpoints,t);
% 2. Compute the vector "f":
        for n = 1:N
            f(n) = -trapz(xpoints,W(n,:).*pvec);
        end
        for n = 1:N
            f(N+n) = trapz(xpoints,Phi(n,:).*pvec*data.c);
        end
% 3. Compute the vector "b":
        b = [zeros(2*N,1); M\f];
% 4. Return the right hand side of the system of first order
    linear ODEs:
        odeRHS = H*q+b;
end
```

This function is used in `ode45` in order to obtain a numerical solution:

```
[T,Q] = ode45(@RHS,tpoints,initial);
```

`Q` is the computed function values for $\mathbf{q}$. Recall from equations (3.93) and (3.61) that `Q` holds the function values for $r_n(t)$ and $s_n(t)$. It also holds function values of its derivatives, but those are of no interest to us. We pick out the solutions for $r_n(t)$ and $s_n(t)$ from `Q` by

```
r = Q(:,1:N);
s = Q(:,N+1:2*N);
```

$r_n(t)$ and $s_n(t)$ are now arrays where the rows correspond to the time values in `tpoints`, which can also be thought of as each row holding the information of a single frame of the animation. The columns correspond to the index n on $r_n(t)$ and $s_n(t)$.

The complete solution is given by (3.97) and (3.98), which are easy to compute as arrays using `ndgrid`. For $w(x,t)$:

```
for i = 1:N
% 1. Convert from vectors to arrays that can be multiplied like
     scalars:
        [WGRID,rGRID] = ndgrid(W(i,:),r(:,i));
% 2. Compute the solution by adding one term of the summation at a
      time:
        w = w + WGRID.*rGRID;
end
```

and likewise for $\phi(x,t)$.

4.2 The layout of the gui

The GUI of the program is seen in Figure 4.1. The GUI is split into a left column holding the two primary buttons and status messages. Input and output are split into separate panels, each featuring tabs to navigate between input options or different output graphics.

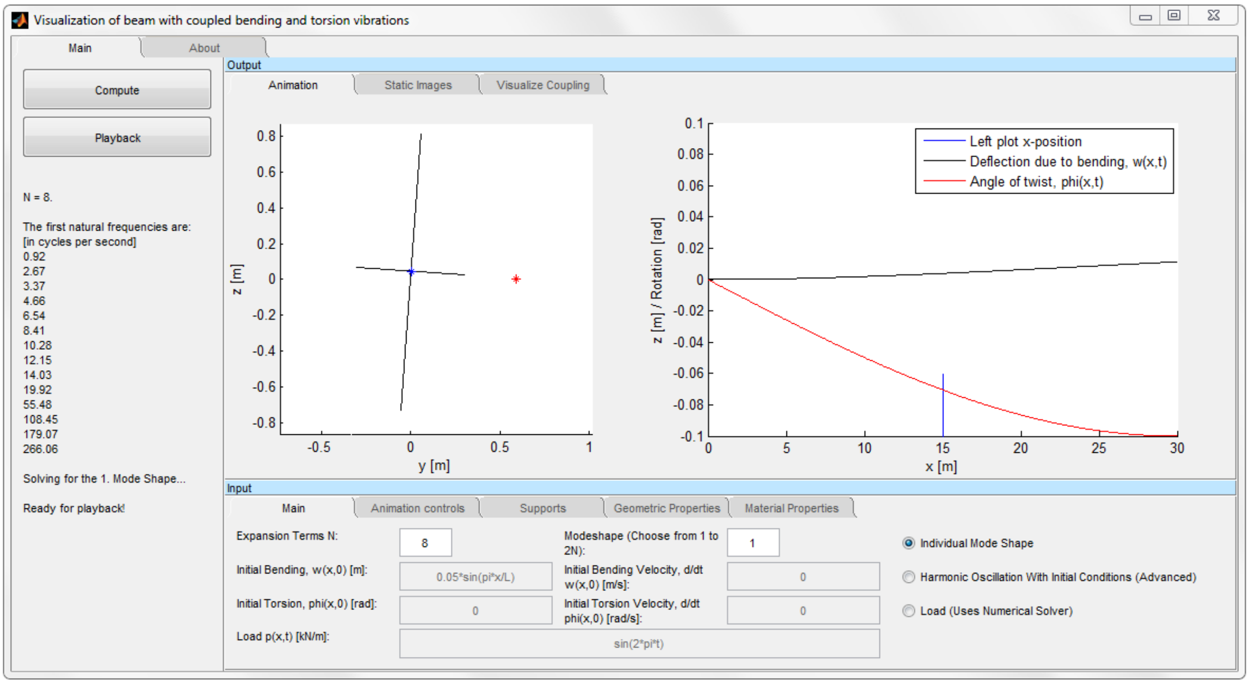


Figure 4.1: The layout of the program's graphical user interface.

The default output window shows the animations. The left animation shows a representation of the beam cross section, depicted by a cross made from the radius of gyration for both axes rather than an actual drawing of the cross section. The cross is placed at the center of mass. The gyration radii are defined by

$$R_y = \sqrt{\frac{I_y}{A}}; \tag{4.2}$$

$$R_z = \sqrt{\frac{I_z}{A}}, \tag{4.3}$$

where A is the area and the second moments of area I_y and I_z have been defined in equation (2.6). The gyration radii provides a good indication of the distribution of material in the cross section, and avoids the problem of having to actually draw the cross section. See Figure 4.2 for an example of how a C-clamp cross section is depicted by the gyration radii.

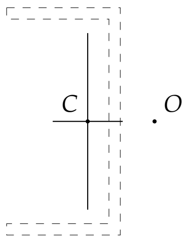


Figure 4.2: Depictions of a C-clamp cross section by the radii of gyration. C is the center of mass and O is the shear center.

This cross moves and rotates during playback of the animation. The x-coordinate used by the left animation is controlled in the `Animation Controls` input tab, entered as a value from zero to unity, where zero corresponds to $x = 0$ and unity corresponds to $x = L$. The position on the axis is illustrated on the right animation with a vertical line.

The right animation displays bending and torsion curves. The horizontal axis has the x-coordinates. It represents the beam length. The bending curve displays the value of $w(x, t)$, that is, the deflection due to bending as illustrated in Figure 2.3. The torsion curve displays the value of $\phi(x, t)$, that is, the twist about the shear center. Again, see Figure 2.3.

The output tab named `Static Images` contains two figures, one for the bending curve and one for the torsion curve. It is not an animation, but an image generated from MATLAB's `imagesc` command, similar to a contour plot. It is there to provide a quick overview of the bending and torsion as functions of x and time t. The columns represent x-coordinates, and the rows represent the frames of the animation. The first frame is at the bottom.

The third output tab, named `Visualize Coupling`, gives insight into the coupling of basis functions. See Figure (4.3). It is essentially aboslute values of the eigenvectors being shown by MATLAB's `imagesc` command. It is all connected together by equations (3.72) and (3.73). The matrix shown consists of the values

$$\begin{bmatrix} |v_{1,1}| & \cdots & |v_{1,2N}| \\ \vdots & \ddots & \vdots \\ |v_{2N,1}| & \cdots & |v_{2N,2N}| \end{bmatrix}. \tag{4.4}$$

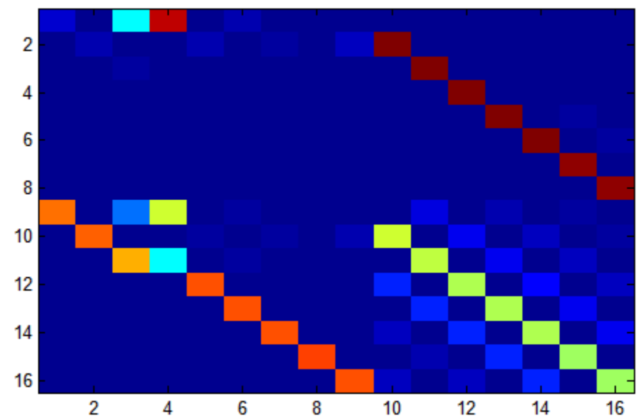


Figure 4.3: An example of the graphic in the `Visualize Coupling` output tab. This is MATLAB's `imagesc` command used on absolute values of the eigenvectors.

A single column corresponds to a single eigenvector and its associated eigenvalue. That means that a single column corresponds to a single natural frequency and mode shape of the vibration. The leftmost column corresponds to the smallest natural frequency. A column will illustrate the coupling of basis functions $W_n(x)$ and $\Phi_n(x)$ for that specific mode shape. The rows correspond to the basis functions and they are ordered this way: The top row is $W_1(x)$. The N'th row from the top is $W_N(x)$. Row number $N+1$ from the top is $\Phi_1(x)$, so it splits between bending and torsion on the middle. The bottom row is Φ_N.

When the user has chosen to visualize a single mode shape, that is essentially a single eigenvector, or equivalently, a single column of (4.4) being used in the animations, as discussed in section (3.3.1). So it is possible to predict from the `Visualize Coupling` tab which basis functions $W(x)$ and $\Phi(x)$ will dominate a particular natural frequency. Or in what proportion bending and torsion occur at a particuar natural frequency. If the distance between the shear center and the center of mass c is set to zero, then the bending and torsion vibrations are not coupled at all. This is directly visible from this graphic, see Figure 4.4. If we gradually increase the value of c, then we also see the coupling becoming gradually more apparent from this graphic.

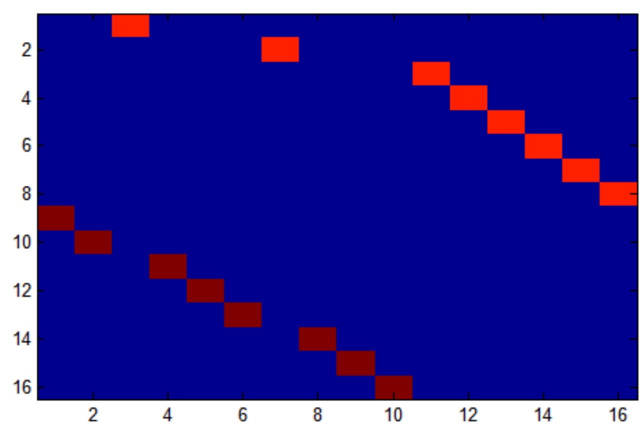


Figure 4.4: When the distance between the shear center and center of mass is set to zero $c = 0$, the system is uncoupled. It is clear from this image that there is no coupling of basis functions.

4.2.1 *Specifying input*

Most of the input fields aims to be rather self explanatory. There is a tab for geometric properties of the beam and its cross section, including such options as the beam length or the distance between the shear center and the center of mass. There is no input for the polar moment of area I_p. Instead there are inputs for I_z, which there would have to be anyway, and an input for I_y. The polar moment of area are computed from these by

$$I_p = I_z + I_y. \tag{4.5}$$

There are also tabs for material properties and for choosing support conditions at the beam ends. And there is an input tab containing parameters for the animation such as the animation duration and framerate, as well as the time range of the animation from $t = 0$ to a value specified by the user.

The main input tab is used to specify the type of problem to visualize. The user may choose between visualizing a single mode shape, the natural response or the forced response. The number of terms N to include in the series expansions equations (3.21) and (3.33) is also specified here. When visualizing a single mode shape, the user is asked to provide a single value to specify which mode shape. That value may be anything from 1 to $2N$, as there are $2N$ natural frequencies and mode shapes.

The initial conditions are entered as functions of x. Incidentally, the beam length L is recognised as the variable `L`. This means for example that if we have selected fixed-fixed supports for torsion, and wanted to say that the initial torsion is a half-period of a sine curve with amplitude 0.1, then the input for $\phi(x, 0)$ may be entered as `0.1*sin(pi*x/L)`. Care must be taken to give reasonable initial conditions, depending on the chosen support conditions! This is not especially easy for the majority of the support types.

When visualizing a forced response, the external load $p(x, t)$ is entered in the same way, only it may also include a temporal variable t. A constant uniformly distributed load would be entered as just a number with no dependence on x or t. At present, the load has to be entered as a function. Consequently, a concentrated load cannot be entered. The reason for this is touched upon in section 4.7.

4.2.2 *The default input*

The default input which fills the input fields when the program is launched matches the cross section shown on Figure 4.5. It is a C-clamp profile. The torsional stiffness parameter K and the position of the shear center and the center of mass, were all computed in the MATLAB program `BeamSec`[2].

At default the beam length is set to 40 meters, and the material parameters correspond to steel.

[2] Jan Becker Høgsberg and Steen Krenk. Analysis of moderately thin-walled beam cross-sections by cubic isoparametric elements. *Computers and Structures*, 134:88–101, 2014. ISSN 0045-7949. DOI: 10.1016/j.compstruc.2014.01.002

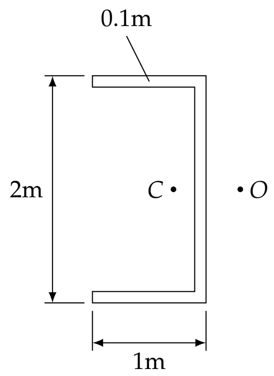


Figure 4.5: The default input values are based on this cross-section.

4.3 *Overview of code files*

The following is an overview of the files that make up the program.

`launcher.m` The only file a user should have to run, as it opens the GUI window. It also adds the necessary directories to the path for the duration of the current session before it calls `opengui.m`.

`opengui.m` Launches an instance of the GUI window. All the GUI elements and in general everything concerning the GUI layout is defined in this function. Apart from its main function, it includes callback functions for the two buttons `Compute` and `Playback`, as well as callback functions for radio button groups.

`defaults.m` Contains default values for the GUI input parameters. When this function is called, the default values are written to the input fields. Calling this function is the last action performed by `opengui` when the GUI is launched, and `defaults` is only called this once. The purpose of separating this functionality into its own function, is to make it very easy to change what default values are used by the program.

`collectinput.m` Collects user input from the input tabs in the GUI, and saves it into `guidata`, a structure used to pass information around between all the functions of the GUI, see section 4.5.

`solver.m` Computes the vibration $w(x,t)$ and $\phi(x,t)$ from the input values. This is the implementation of the results from chapter 3.

`plotting.m` Turns the solution obtained by `solver` into graphics and animations, and updates the output tabs of the GUI.

`playback.m` Plays the animation. An entire function has been written for the purpose of animating two animations at the same time, as no built in MATLAB command supports this. This function is described in section 4.6.

`notify.m` Updates the leftmost panel of the GUI with information to the user. This function is called several times from within other functions, in order to update the panel with the status of the GUI.

When the program is launched by calling `launcher`, this in turn calls `opengui.m`, which calls `defaults.m` as its last action. See Figure (4.6). Most of the other files are activated when the `Compute` button is pressed. The process set in motion by the `Compute` button is illustrated in Figure 4.7. The button click is first picked up by its callback function defined in `opengui`. The callback function then calls `collectinput`, `solver` and `plottng` in succession. The `Playback` button only calls `playback.m`.

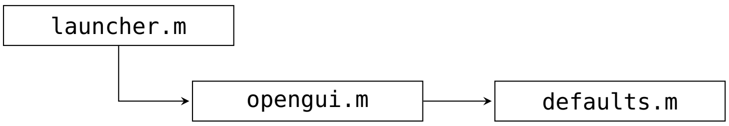


Figure 4.6: When the software is launched, `opengui.m` is called, which in turn calls `defaults.m`.

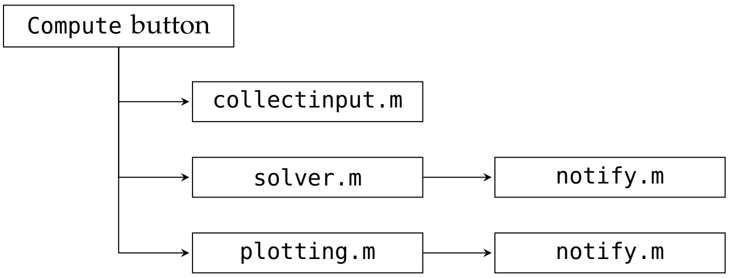


Figure 4.7: The `Compute` button calls three functions, `collectinput`, `solver` and `plotting`. During the computation, several calls are made to `notify` as well.

4.4 *The MATLAB Layout Toolbox by The MathWorks Ltd*

The MATLAB graphical user interface or GUI is built using a small toolbox called the MATLAB Layout Toolbox by The MathWorks Ltd. This toolbox is not a native part of MATLAB, and has to be loaded onto the path separatedly. The necessary files are included with the MATLAB files for the program handed in as part of this bachelor's project.

The toolbox lets the user arrange GUI elements in much the same way as HTML does, by nesting bodies of content within other bodies and distributing elements either horizontally or vertically. It is a small collection of rather simple layout primitives, which offer great flexibility when combined.

The advantages of using this over MATLAB's built in GUIDE tool for creating GUIs, is that it allows precise arrangement in a source format that is entirely text based and human readable. In contrast to this, creating GUIs with GUIDE is mostly a drag and drop kind of workflow, not one of manually writing code. The downside to the GUIDE approach, besides the difficulties concerning precise arrangement, is that the code created by MATLAB, describing the GUI layout, is contained in a .fig file format which is not human readable. The GUI in this project was build in part as a learning experience. As I had no prior experience with MATLAB, being able to see the gears and wheels and inner workings of the code comprising the GUI was much appreciated. In addition, the software is intended as a tool for others to use and possibly modify.

Therefore, it was seen as a great advantage to have the code be human readable and working with the Layout Toolbox has been an excellent experience.

4.5 *Passing data and handles between functions*

The GUI is made up of mostly dynamic elements and of few static ones. Dynamic elements are any elements that changes its content at some point. Examples are input fields, output graphics, radio buttons and user messages. (radio buttons groups of round buttons where only one may be selected at any time). Any `uicontrol` element (which is a typical MATLAB GUI element like a button) created with a `Tag` property automatically receives a handle by MATLAB. These handles are retrieved by calling the `guihandles` function. However, some elements; the axes and the implementation of the radio buttons, are not native GUI elements and are therefore not compatible with the `guihandles` structure. The handles to these elements are static, but are instead assigned specifically and stored in `guidata`. A structure used to store dynamic data in the GUI and pass it around between functions.

`guidata` is used to store any data passed between different functions. In essence, a function will first load the current data from `guidata`, then perform its purpose with that data, and finally store its results by updating `guidata` with any changes. Taking `solver.m` as an example, the function first loads the data from `guidata` in order to get the user input, which has been put into `guidata` by `collectinput.m` after the `Compute` button has been clicked. It then computes solutions and stores them back into `guidata` for `plotting.m` to use next.

When data is loaded at the start of a function through `guidata`, it is loaded into the variable `data` and the actual data is stored in fields. Example: `data.L` is the beam length.

4.6 *Rendering and playing back the animations*

Since MATLAB's built-in animation function `movie` can only handle one animation at a time, it was necessary to write a complete function `playback.m` to play back two simultaneous animations. The way in which this is achieved also affects the way in which the animations are pre-rendered in `plotting.m`.

To achieve two simultaneous animations, the frames of an animation are not grapped as snapshots of the axes as is the case when using `movie` with `getframe`. Instead the solution to the problem conceived in this project, is to render every line or point of every frame of the animation into the same axes by multiple uses of the `plot` command. Handles to these layers of lines and points are saved and stored into `guidata`, which allow them to be turned on and off, and that is what `playback.m` does. An animation is really rendered to look like Figure 4.8, but never are several layers visible at the same time.

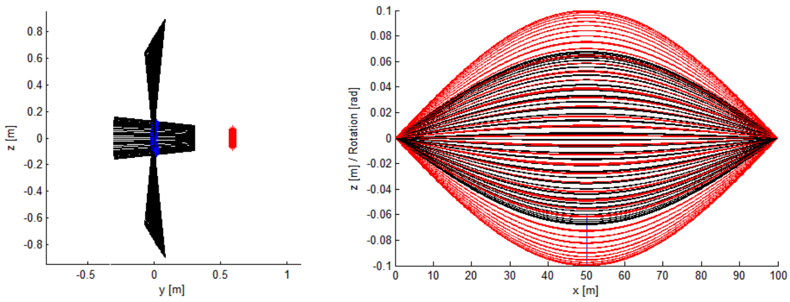


Figure 4.8: In order to achieve simultaneously playing animations, an animation is rendered as a single plot, but with its content sorted into layers that can be switched on and off. This shows an animation rendered with all layers turned on.

In order to render all lines and points into the same set of axes, the `hold` property of the axes are set to on, which makes MATLAB draw on top of the existing content on the axes as the frames are rendered:

```
hold(data.animationleft,'on');
hold(data.animationright,'on');
```

`data.animationleft` and `data.animationright` are handles to the axes. Handles to every `plot` command of used in the rendering of each frame are then saved into one dimensional arrays. Taking the right animation as an example, this is achieved in the following loop:

```
for frame=1:data.nframes
    %bending curve
    y = data.w(:,frame);
    layerBendingCurve(frame) = plot(data.animationright,data.
        xpoints,y,'k','visible','off');

    %twist curve
    y = data.phi(:,frame);
    layerTorsionCurve(frame) = plot(data.animationright,data.
        xpoints,y,'r','visible','off');
end
```

The arrays `layerBendingCurve` and `layerTorsionCurve` hold handles to the curves of the right plot. Inside `playback.m`, which loops over the frames, these are then switched on and off by the following code:

```
% Turn off previous frame:
  set(data.layerBendingCurve(frame-1),'visible','off');
  set(data.layerTorsionCurve(frame-1),'visible','off');
% Turn on current frame:
  set(data.layerBendingCurve(frame),'visible','on');
  set(data.layerTorsionCurve(frame),'visible','on');
```

The above few lines are just an example to illustrate how the animation is handled. The above are only a part of a larger loop in the actual code, since the loop in `playback.m` also has to perform some checks and actions at the start and end of the loop. See the full code in the appendix section B.7.

4.7 Notes on the development process

As any novice in MATLAB with little knowledge of the language is prone to do, `solver.m` was first build using anonymous functions for many expressions. For example the basis functions $W_n(x)$ and $\Phi_n(x)$ were defined as anonymous functions. An anonymous function in MATLAB would look like e.g.

```
f = @(x) sin(x)
```

Down through the code, other functions or expressions were defined on top of the previously defined anonymous functions. Integrations were performed by the MATLAB `integral` command. Ultimately a triple loop would compute the numerical values of the solution used by `plotting.m` to generate the animations. This triple loop would loop over every x value (approximately 500 values, one value for each pixel on the x axis of the animation), every t value which is the number of frames in the animation, and a third parameter which looped through terms in the sums of equation (3.72) and (3.73). The complete `solver` function was implemented like this, and the software worked just fine except for being painfully slow, requiring the user to wait 8-10 seconds each time the `Compute` button was pressed. These speed issues were seen as problematic for a program which is meant to be played with by students, where the learning comes from repeatedly trying different input parameters. As it became apparent that the approach taken was not catering to MATLAB's strengths, the whole `solver` function was rewritten, resulting in a massive improvement in evaluation time. The result is that instead of anonymous functions, the intermediary results are kept as just lots of numerical values in arrays. Use of MATLAB's `integral` function was replaced by numerical integrations on arrays performed by `trapz`, and costly loops were replaced by matrix manipulations.

The advantages of the first approach were that the code more closely resembled the equations in chapter 3. Also the use of anonymous functions allowed things like the dirac delta to be used when specifying an external load, as this is handled correctly by MATLAB's `integral` function. With the way the program currently works (by using arrays of numerical points with some set resolution instead of anonymous functions), it would require a little rewriting to allow a diraq delta to be used in specifying an external load. The consequence is that a concentrated external force cannot currently be entered. The speed increase is well worth it though.

The GUI was also first build as a single tab including all input fields and output. As the number of input options gradually grew, the GUI started to become relatively large and confusing at first sight. This led to rebuilding the GUI from scratch, now allowing input fields and output graphics to be split into tabs, as well as gaining a left margin used to display status messages.

4.8 Modification to the Layout Toolbox

A single parameter was changed in one of the files of the Layout Toolbox used when designing the GUI in order to allow wider tabs. Line 29 was changed in `TabPanel.m` of the Layout Toolbox from its default value of 50 to 110:

```
28 properties
29     TabSize = 110
30     TabPosition = 'top'
31 end
```

4.9 Known bugs

The output from MATLAB's `eig` function sometimes suddenly becomes unpredictable. This bug has been observed to appear once the largest eigenvalue is above 10^8, corresponding to natural frequencies of more than 1500 cycles per second. The program tries to detect this bug by looking for negative eigenvalues, which seem to accompany this bug. When it is detected, an exception is thrown and a message is displayed with a warning. The user can then modify the input and try again.

This sometimes happen when a large N is specified, because it causes a large number of eigenvalues to be computed. The largest eigenvalues apparently become too much to handle. This looks like a bug in MATLAB's `eig` command. Whether it is or not, I am not yet certain. However, as the `eig` command is copyrighted by MathWorks and its source cannot be viewed, and since the program heavily depends on the use of `eig`, this bug has not been fixed yet.

4.10 Further development

It would be interesting to do error analysis on the solutions found for the coupled equations of motion. Most notably, what is the effect of increasing the number of terms N in the expansion? At what angle of twist is the assumption of linearity no longer viable? Carrying out this error analysis has not been prioritized because the purpose of the solutions and the software is to give students intuitive feel for the vibrations. It is of minor importance to this purpose whether the solution is a few percent off target. However, a warning *is* displayed to the user if the angle of twist computed for the animation reaches above 0.1 radians.

It has been the intention to include a `save` button to save the current input. Or even to save input automatically so the program would load the input of the last session at launch. Accompanyed by a `Reset to defaults` button this would be a nice feature to have. As it stands, the user loses all input when the program is closed. It is easy to edit `defaults.m` by hand, but that is not quite the same. This save feature would not be difficult to implement by using a function similar to `defaults.m`, however it all takes time, and this feature remains an idea for further development.

A complete function has been written to export the animations to movie files. This feature was a biproduct of searching for a way to show two simultaneous animations, it was not written because the ability to export movies has been prioritized. However it works perfectly except labels and tick marks are not included in the exported movie. Because this bug has not been easy to fix, the ability to export animations have ultimately not been included in the program. The code is still there among the other code files in the form of the file `exportanimation.m`.

At present it is a little difficult to specify initial conditions for the natural response, as they have to be specified as functions. This does have utility, as it represents freedom, but it might be nice to add functionality that allows the user to select certain basis functions which are then excited and used as initial conditions.

Another inviting idea for visualization is to have the beam drawn in 3D and animate that. There is no *need* to draw the actual cross section, as the representation by the gyration radii could also be used in 3D. The vibration results from this program could potentially be exported into another program that does this, even with an actual drawing of the cross section.

The input of especially the material properties might be done by dropdown menues with pre-defined materials.

A
Appendix

A.1 Relation between moment of inertia and polar moment of area

The centroidal moment of inertia I_{CM} about an axis parallel to the x axis of Figure 2.2 and going through the center of mass is defined as $I_m = \int_V \rho(r) r^2 \, \mathrm{d}V$. r is the distance from a point to the axis and V is the volume. The polar moment of area about the same axis is $I_p = \int_A r^2 \, \mathrm{d}A$.

If the cross-section is homogenous, then the density ρ is a constant. For a uniform beam segment of length dx, the distance r does not depend on x. This leads to

$$\begin{aligned} I_{CM} &= \int_V \rho(r) r^2 \, \mathrm{d}V \\ &= \rho \int_V r^2 \, \mathrm{d}V \\ &= \rho \int_A \int_x^{x+dx} r^2 \, \mathrm{d}x \, \mathrm{d}A \\ &= \rho \, dx \int_A r^2 \, \mathrm{d}A \\ &= \rho \, dx \, I_p. \end{aligned} \tag{A.1}$$

A.2 Orthogonality conditions

Four conditions of orthogonality are shown below. Two for bending basis functions, and two for torsion basis functions. Taking (3.5) as a starting point, take two equations of differing indices

$$W_n'''' - \alpha_n^4 W_n = 0; \tag{A.2}$$

$$W_k'''' - \alpha_k^4 W_k = 0. \tag{A.3}$$

Multiply by W_k and W_n respectively and integrate over the beam length to get

$$\int_0^L W_k W_n'''' \, \mathrm{d}x - \alpha_n^4 \int_0^L W_k W_n \, \mathrm{d}x = 0; \tag{A.4}$$

$$\int_0^L W_n W_k'''' \, \mathrm{d}x - \alpha_k^4 \int_0^L W_n W_k \, \mathrm{d}x = 0. \tag{A.5}$$

Integration by parts yields

$$\int_0^L W_k'' W_n''\,\mathrm{d}x - \alpha_n^4 \int_0^L W_k W_n\,\mathrm{d}x = 0; \tag{A.6}$$

$$\int_0^L W_n'' W_k''\,\mathrm{d}x - \alpha_k^4 \int_0^L W_n W_k\,\mathrm{d}x = 0, \tag{A.7}$$

where the byproducts $[W_k W_n''']_0^L$ and $[W_k' W_n'']_0^L$ are not written, as the boundary conditions forces them to be equal to zero. Subtracting the equations from each other leaves

$$\left(\alpha_k^4 - \alpha_n^4\right) \int_0^L W_k W_n\,\mathrm{d}x = 0. \tag{A.8}$$

As α^4 describes the spatial frequency of the basis function, $k \neq n$ implies $\alpha_k^4 \neq \alpha_n^4$, which further implies that

$$\int_0^L W_k W_n\,\mathrm{d}x = 0, \quad \text{for } k \neq n. \tag{A.9}$$

This is the first condition of orthogonality for the bending basis functions.

Divide (A.6) and (A.7) by α_n^4 and α_k^4 respectively to get

$$\frac{1}{\alpha_n^4} \int_0^L W_k'' W_n''\,\mathrm{d}x - \int_0^L W_k W_n\,\mathrm{d}x = 0; \tag{A.10}$$

$$\frac{1}{\alpha_k^4} \int_0^L W_n'' W_k''\,\mathrm{d}x - \int_0^L W_n W_k\,\mathrm{d}x = 0. \tag{A.11}$$

Subtract the equations from each other to get

$$\left(\frac{1}{\alpha_n^4} - \frac{1}{\alpha_k^4}\right) \int_0^L W_k'' W_n''\,\mathrm{d}x = 0. \tag{A.12}$$

Then

$$\int_0^L W_k'' W_n''\,\mathrm{d}x = 0, \quad \text{for } k \neq n, \tag{A.13}$$

which is the second condition of orthogonality for the bending basis functions.

Now, beginning from (3.25) and writing two equations;

$$\Phi_n'' + \beta_n^2 \Phi_n = 0; \tag{A.14}$$

$$\Phi_k'' + \beta_k^2 \Phi_k = 0, \tag{A.15}$$

multiply by Φ_k and Φ_n respectively, integrate along the beam length and perform integration by parts to get

$$-\int_0^L \Phi_k' \Phi_n'\,\mathrm{d}x + \beta_n^2 \int_0^L \Phi_k \Phi_n\,\mathrm{d}x = 0; \tag{A.16}$$

$$-\int_0^L \Phi_n' \Phi_k'\,\mathrm{d}x + \beta_k^2 \int_0^L \Phi_n \Phi_k\,\mathrm{d}x = 0. \tag{A.17}$$

Subtracting these equation from each other, or first dividing by β_n and β_k respectively, and then subtracting them from each other leads to the two equations

$$\left(\beta_k^2 - \beta_n^2\right) \int_0^L \Phi_k \Phi_n \, \mathrm{d}x = 0; \tag{A.18}$$

$$\left(\frac{1}{\beta_n^2} - \frac{1}{\beta_k^2}\right) \int_0^L \Phi_k' \Phi_n' \, \mathrm{d}x = 0. \tag{A.19}$$

This implies that

$$\int_0^L \Phi_k \Phi_n \, \mathrm{d}x = 0, \quad \text{for } k \neq n; \tag{A.20}$$

$$\int_0^L \Phi_k' \Phi_n' \, \mathrm{d}x = 0, \quad \text{for } k \neq n. \tag{A.21}$$

Which are the orthogonality conditions for the torsion basis functions.

B
Code

B.1 launcher.m

```
%% LAUNCHER
% This file adds code directories to the path and launches the program GUI.

%-----------------------------------------------------------------------%
%% Add code to the path.
% This is not permanent - it disappears after the current session.
thisdir = fileparts( mfilename( 'fullpath' ) );
fprintf('Adding the following directories to the path for the duration of the current session:\n');
dirs = {
    fullfile( thisdir)
    fullfile( thisdir, 'GUITools' )
    fullfile( thisdir, 'GUITools', 'Patch' )
    fullfile( thisdir, 'code' )
    };
for dd=1:numel( dirs )
    addpath( dirs{dd} );
    fprintf( '+ %s\n', dirs{dd} );
end

%-----------------------------------------------------------------------%
%% Open GUI:
opengui()
```

B.2 opengui.m

```
function opengui
% This will open the GUI window.
% The contents of this file both constructs and arranges elements inside a
% GUI window. At the bottom of this file are callback functions for GUI
% elements, which determines actions taken by the GUI when the user
% interacts with it.
%
% This GUI is built using GUI Layout Toolbox version 1.17 from MathWorks
% Ltd.
% This makes building the GUI similar to HTML/CSS layout, by allowing
% nested elements as well as layout by specifying element properties such as
% padding.

efh = 27; % A constant used throughtout for the vertical height of input fields. efh is for EditableFieldHeight
p = 10; % constant for padding
s = 5; % constant for spacing

%------------------------------------------------------------------------%
%% Open GUI window
window = figure( ...
    ’Name’, ’Visualization␣of␣beam␣with␣coupled␣bending␣and␣torsion␣vibrations’, ...
    ’Position’, [70 100 1160 600], ...
    ’MenuBar’, ’none’, ...
    ’Toolbar’, ’none’, ...
    ’NumberTitle’, ’off’);

%------------------------------------------------------------------------%
%% Setup tabbed layout
% Create the horizontal panel which holds the tabs:
tabs = uiextras.TabPanel( ...
    ’Parent’, window );
% Create main tab, horizontally distributed:
maintab = uiextras.HBox( ...
    ’Parent’, tabs);
% Other tabs are created later, once this main tab has been filled

%------------------------------------------------------------------------%
%% Create left window with updating text and buttons
% Create a column for holding the pushbuttons and the message box:
leftcolumn = uiextras.VBox( ...
    ’Parent’, maintab, ...
    ’Padding’, p, ...
    ’Spacing’, s);
% Create a button for preparing the animation:
uicontrol( ...
    ’Parent’, leftcolumn, ...
    ’Style’, ’pushbutton’, ...
    ’String’, ’Compute’, ...
    ’Tag’,’computebutton’, ...
    ’Callback’,@compute);
% Create a for playing back the animation:
uicontrol( ...
    ’Parent’, leftcolumn, ...
    ’Style’, ’pushbutton’, ...
    ’String’, ’Playback’, ...
    ’Enable’, ’Off’, ... % Disable until animation is prepared.
    ’Tag’,’playbackbutton’, ...
    ’Callback’,@playbackanimation);
% Create a line of text for showing the value of t while the animation runs.
% This cannot be displayed as part of the larger box of text below,
% because this is updated for every frame during playback, and it is too
% intensive so the playback will lag a lot, even on a decent pc. But it works
% ok with its own box.
uicontrol( ...
    ’Parent’, leftcolumn, ...
```

```
    'Style','text', ...
    'Tag','animationtime');
% Create the message box which holds info to the user. This is updated by
% subfunctions to reflect the current state of the program:
uicontrol( ...
    'Parent', leftcolumn, ...
    'Style','text', ...
    'Tag','console', ...
    'HorizontalAlignment','left');
% Set a fixed height of buttons:
set(leftcolumn, 'Sizes', [40 40 20 -1] )

%----------------------------------------------------------------------%
%% Create Output and Input panels
% Hold the panels in a vertical box:
panelvbox = uiextras.VBox( ...
    'Parent', maintab);
% by now the maintab has been fully filled, so define its spacing
% distribution:
set(maintab, 'Sizes', [200 -1]) % [leftcolumn panelvbox] in pixels. -1 means auto scale with the window.
% Create output panel:
outputpanel = uiextras.BoxPanel( ...
    'Parent', panelvbox, ...
    'Title', 'Output');
outputpaneltabs = uiextras.TabPanel( ...
    'Parent',outputpanel);
% Create input panel:
inputpanel = uiextras.BoxPanel( ...
    'Parent', panelvbox, ...
    'Title', 'Input');
inputpaneltabs = uiextras.TabPanel( ...
    'Parent',inputpanel);
set(panelvbox, 'Sizes', [-1 180])

%----------------------------------------------------------------------%
%% Fill output panel animation tab
% Create horizontal box that holds the animation
outputanimationhbox = uiextras.HBox( ...
    'Parent', outputpaneltabs, ...
    'Padding', 0, ...
    'Spacing', 0);
% Create axes for the gyration radius animation (left):
% I found it necessary to create a fresh VBoc or HBox to hold JUST the axes
% when using the 'OuterPosition'. This is likely a bug in the GUI Layout Toolbox,
% but this little hack gets around it. Without this, the spacing and padding is
% way off.
animationleftcontainer = uiextras.VBox( ...
    'Parent', outputanimationhbox);
data.animationleft = axes( ...
    'Parent', animationleftcontainer, ...
    'ActivePositionProperty', 'OuterPosition');
% Create axes for plotting the bending/torsion animation (right):
animationrightcontainer = uiextras.VBox( ...
    'Parent', outputanimationhbox);
data.animationright = axes( ...
    'Parent', animationrightcontainer, ...
    'ActivePositionProperty', 'OuterPosition');
% Allocate a fixed width to the left plot, and let the right plot scale
% with the window
set(outputanimationhbox, 'Sizes', [-1 -1.5] )

%----------------------------------------------------------------------%
%% Fill output panel static tab
outputstatichbox = uiextras.HBox( ...
    'Parent', outputpaneltabs, ...
    'Padding', p, ...
    'Spacing', 2*p);
% Create static text
```

```
text = sprintf('\nLEFT:␣Bending.\nRIGHT:␣Torsion.\n\nThis␣provides␣a␣quick␣overview␣of␣the␣vibration.␣It␣contains␣the␣same␣information␣as␣the␣animation.\n\nThe␣left␣plot␣shows␣deflection␣due␣bending␣and␣the␣right␣shows␣torsional␣deflection.␣The␣horizontal␣axes␣spans␣the␣x␣coordinates␣from␣0␣to␣L.␣The␣vertical␣axes␣are␣time␣with␣t=0␣at␣the␣bottom.␣\n\nThis␣is␣the␣standard␣MATLAB␣color␣scale,␣from␣dark␣blue␣to␣dark␣red,␣and␣light␣green␣as␣the␣middle␣value.␣Go␣to␣the␣animation␣tab␣to␣read␣the␣actual␣values␣off␣the␣axes,␣as␣the␣color␣scale␣is␣not␣displayed␣here.');
uicontrol( ...
    'Parent', outputstatichbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String',text);
% Create axes for bending (left):
data.staticaxesbending = axes( ...
    'Parent', outputstatichbox, ...
    'ActivePositionProperty', 'Position', ...
    'HandleVisibility', 'Callback' );
set(data.staticaxesbending,'XTickLabel','','YTickLabel','')
% Create axes for torsion (right):
data.staticaxestorsion = axes( ...
    'Parent', outputstatichbox, ...
    'ActivePositionProperty', 'Position', ...
    'HandleVisibility', 'Callback');
set(data.staticaxestorsion,'XTickLabel','','YTickLabel','')
set(outputstatichbox,'Sizes',[200, -1 -1])

%-----------------------------------------------------------------------%
%% Fill output panel coupling tab
outputcouplinghbox = uiextras.HBox( ...
    'Parent', outputpaneltabs, ...
    'Padding', p, ...
    'Spacing', s);
% Create static text:
text = sprintf('\nThis␣shows␣the␣coupling␣of␣bending␣and␣torsion␣basis␣functions␣for␣individual␣mode␣shapes.\n\nEach␣column␣represents␣a␣mode␣shape␣corresponding␣to␣a␣natural␣frequency.␣The␣first␣mode␣shape,␣corresponding␣to␣the␣lowest␣natural␣frequency,␣appears␣in␣the␣leftmost␣column,␣and␣the␣last␣modeshape␣appears␣in␣the␣rightmost␣column.\n\nThe␣rows␣represent␣basis␣functions␣in␣the␣series␣expansion,␣with␣the␣upper␣half␣representing␣bending␣basis␣functions,␣and␣the␣lower␣half␣representing␣torsion␣basis␣functions.\n\nThis␣shows␣absolute␣values,␣in␣a␣standard␣MATLAB␣color␣scale,␣where␣the␣range␣is␣from␣dark␣blue␣to␣dark␣red.\n\nTo␣get␣a␣feel␣for␣it,␣try␣an␣uncoupled␣system␣by␣setting␣c=0␣(the␣shear␣center␣to␣elastic␣center␣distance␣under␣the␣Geometric␣Properties␣tab)');
uicontrol( ...
    'Parent', outputcouplinghbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String', text);
uiextras.Empty( ...
    'Parent',outputcouplinghbox);

% Create axes:
test = uiextras.VBox( ...
    'Parent', outputcouplinghbox);
data.staticaxescoupling = axes( ...
    'Parent', test, ...
    'ActivePositionProperty', 'OuterPosition', ...
    'HandleVisibility', 'Callback');
set(data.staticaxescoupling,'XTickLabel','','YTickLabel','')
set(outputcouplinghbox, 'Sizes', [250 -1.3 -5] )

outputpaneltabs.TabNames = {'Animation', 'Static␣Images', 'Visualize␣Coupling'};
outputpaneltabs.SelectedChild = 1; % Open the program with the first tab active

%-----------------------------------------------------------------------%
%% Load, initial conditions and modeshapes.
maininputtab = uiextras.HBox( ...
    'Parent', inputpaneltabs, ...
    'Padding', p, ...
    'Spacing', 20);
maininputtabvbox = uiextras.VBox( ...
```

```
    'Parent', maininputtab, ...
    'Spacing', s);
maininputtabhbox = uiextras.HBox( ...
    'Parent', maininputtabvbox, ...
    'Spacing', 10);
% Disstribute into columns:
maininputcolumn1 = uiextras.VBox( ...
    'Parent', maininputtabhbox, ...
    'Spacing', s);
maininputcolumn2 = uiextras.VBox( ...
    'Parent', maininputtabhbox, ...
    'Spacing', s);
set(maininputtabhbox, 'Sizes', [300 300] )
%% Create an input for N:
Nhbox = uiextras.HBox( ...
    'Parent', maininputcolumn1, ...
    'Spacing', s);
uicontrol( ...
    'Parent', Nhbox, ...
    'Style','text', ...
    'HorizontalAlignment', 'left', ...
    'String','Expansion␣Terms␣N:');
uicontrol( ...
    'Parent', Nhbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','N');
set(Nhbox, 'Sizes', [150 50] )
%% Create an input for selecting modeshape:
modeshapehbox = uiextras.HBox( ...
    'Parent', maininputcolumn2, ...
    'Spacing', s);
uicontrol( ...
    'Parent', modeshapehbox, ...
    'Style','text', ...
    'HorizontalAlignment', 'left', ...
    'String','Modeshape␣(Choose␣from␣1␣to␣2N):');
uicontrol( ...
    'Parent', modeshapehbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','modeshape');
set(modeshapehbox, 'Sizes', [150 50] )

%% Create inputs for initial conditions:
initialwhbox = uiextras.HBox( ...
    'Parent', maininputcolumn1, ...
    'Spacing', s);
uicontrol( ...
    'Parent', initialwhbox, ...
    'Style','text', ...
    'HorizontalAlignment', 'left', ...
    'String','Initial␣Bending,␣w(x,0)␣[m]:');
uicontrol( ...
    'Parent', initialwhbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','initialw', ...
    'Enable','off');
set(initialwhbox, 'Sizes', [150 -1] )
initialphihbox = uiextras.HBox( ...
    'Parent', maininputcolumn1, ...
    'Spacing', s);
uicontrol( ...
    'Parent', initialphihbox, ...
```

```
    'Style','text', ...
    'HorizontalAlignment', 'left', ...
    'String','Initial␣Torsion,␣phi(x,0)␣[rad]:');
uicontrol( ...
    'Parent', initialphihbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','initialphi', ...
    'Enable','off');
set(initialphihbox, 'Sizes', [150 -1] )
initialwdothbox = uiextras.HBox( ...
    'Parent', maininputcolumn2, ...
    'Spacing', s);
uicontrol( ...
    'Parent', initialwdothbox, ...
    'Style','text', ...
    'HorizontalAlignment', 'left', ...
    'String','Initial␣Bending␣Velocity,␣d/dt␣w(x,0)␣[m/s]:');
uicontrol( ...
    'Parent', initialwdothbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','initialwdot', ...
    'Enable','off');
set(initialwdothbox, 'Sizes', [150 -1] )
initialphidothbox = uiextras.HBox( ...
    'Parent', maininputcolumn2, ...
    'Spacing', s);
uicontrol( ...
    'Parent', initialphidothbox, ...
    'Style','text', ...
    'HorizontalAlignment', 'left', ...
    'String','Initial␣Torsion␣Velocity,␣d/dt␣phi(x,0)␣[rad/s]:');
uicontrol( ...
    'Parent', initialphidothbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','initialphidot', ...
    'Enable','off');
set(initialphidothbox, 'Sizes', [150 -1] )

set(maininputcolumn1, 'Sizes', [efh efh efh] )
set(maininputcolumn2, 'Sizes', [efh efh efh] )

%% Create an input for specifying load:
loadhbox = uiextras.HBox( ...
    'Parent', maininputtabvbox, ...
    'Spacing', s);
uicontrol( ...
    'Parent', loadhbox, ...
    'Style','text', ...
    'HorizontalAlignment', 'left', ...
    'String','Load␣p(x,t)␣[kN/m]:');
uicontrol( ...
    'Parent', loadhbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','load', ...
    'Enable','off');
set(loadhbox, 'Sizes', [150 -1] )
set(maininputtabvbox, 'Sizes', [3*efh+2*s efh] )

%% Radiobuttons which chooses the solver that's used
maininputradiobuttons = uiextras.VBox( ...
```

```
    'Parent', maininputtab, ...
    'Spacing', s);
data.solverhandle(1) = uicontrol(...
    'Parent', maininputradiobuttons, ...
    'Style', 'radiobutton', ...
    'Callback', @radiosolver, ...
    'String',   'Individual␣Mode␣Shape', ...
    'Value',    1); % set the whole solution as default
data.solverhandle(2) = uicontrol(...
    'Parent', maininputradiobuttons, ...
    'Style', 'radiobutton', ...
    'Callback', @radiosolver, ...
    'String',   'Harmonic␣Oscillation␣With␣Initial␣Conditions␣(Advanced)', ...
    'Value',    0);
data.solverhandle(3) = uicontrol(...
    'Parent', maininputradiobuttons, ...
    'Style', 'radiobutton', ...
    'Callback', @radiosolver, ...
    'String',   'Load␣(Uses␣Numerical␣Solver)', ...
    'Value',    0);
set(maininputradiobuttons, 'Sizes', [efh efh efh] )
set(maininputtab, 'Sizes', [610 -1] )

%----------------------------------------------------------------------%
%% Input panel animation controls tab
animationcontroltab = uiextras.HBox( ...
    'Parent', inputpaneltabs, ...
    'Padding',p);
% Create a vertical box to hold input fields
animationcontrolvbox = uiextras.VBox( ...
    'Parent', animationcontroltab, ...
    'Spacing',s);
set(animationcontroltab, 'Sizes', [300] )
%% Create input for the x-coordinate of the gyration radius plot:
xcoorhbox = uiextras.HBox( ...
    'Parent', animationcontrolvbox, ...
    'Spacing', s);
uicontrol( ...
    'Parent', xcoorhbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','x-Coordinate␣For␣The␣Left␣Animation␣(From␣0␣to␣1):');
uicontrol( ...
    'Parent', xcoorhbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','xGyration');
set(xcoorhbox, 'Sizes', [150 100] )
%% Create an input for FPS:
frameratehbox = uiextras.HBox( ...
    'Parent', animationcontrolvbox, ...
    'Spacing', s);
uicontrol( ...
    'Parent', frameratehbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Animation␣framerate␣[fps]:');
uicontrol( ...
    'Parent', frameratehbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','fps');
set(frameratehbox, 'Sizes', [150 100] )
%% Create an input for duration:
durationhbox = uiextras.HBox( ...
    'Parent', animationcontrolvbox, ...
```

```
    'Spacing', s);
uicontrol( ...
    'Parent', durationhbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Animation␣duration␣[s]:');
uicontrol( ...
    'Parent', durationhbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','duration');
set(durationhbox, 'Sizes', [150 100] )
%% Create an input for time range:
timerangehbox = uiextras.HBox( ...
    'Parent', animationcontrolvbox, ...
    'Spacing', s);
uicontrol( ...
    'Parent', timerangehbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Time␣range,␣t␣=␣0␣to␣[s]:');
uicontrol( ...
    'Parent', timerangehbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','tmax');
set(timerangehbox, 'Sizes', [150 100] )
set(animationcontrolvbox, 'Sizes', [efh efh efh efh] )

%-----------------------------------------------------------------------%
%% Input panel support tab
supporttab = uiextras.HBox( ...
    'Parent', inputpaneltabs, ...
    'Padding', p, ...
    'Spacing', 2*s);
bendingBCpanel = uiextras.Panel( ...
    'Parent', supporttab, ...
    'Padding', p, ...
    'Title', 'Bending␣Boundary␣Conditions');
torsionBCpanel = uiextras.Panel( ...
    'Parent', supporttab, ...
    'Padding', p, ...
    'Title', 'Torsion␣Boundary␣Conditions');
% Distribute into columns:
bendingpanelhbox = uiextras.HBox( ...
    'Parent', bendingBCpanel);
supportcolumn1 = uiextras.VBox( ...
    'Parent', bendingpanelhbox);
supportcolumn2 = uiextras.VBox( ...
    'Parent', bendingpanelhbox);
supportcolumn3 = uiextras.VBox( ...
    'Parent', torsionBCpanel, ...
    'Spacing', s);
% Reverse buttons:
reversebuttons = uiextras.VBox( ...
    'Parent', supporttab);
set(supporttab, 'Sizes', [300 150 -1] )

uicontrol(...
    'Parent', reversebuttons, ...
    'Style', 'checkbox', ...
    'Tag', 'reversebending', ...
    'String', 'Reverse␣Bending␣Boundary␣Conditions');
uicontrol(...
    'Parent', reversebuttons, ...
    'Style', 'checkbox', ...
```

```
    'Tag', 'reversetorsion', ...
    'String', 'Reverse␣Torsion␣Boundary␣Conditions');
set(reversebuttons, 'Sizes', [68 15] )

data.bendingBChandle(1) = uicontrol(...
    'Parent', supportcolumn1, ...
    'Style', 'radiobutton', ...
    'Callback', @radiobending, ...
    'String',   'Hinged␣-␣Hinged', ...
    'Value',    1); % set the simple support to default
data.bendingBChandle(2) = uicontrol(...
    'Parent', supportcolumn1, ...
    'Style', 'radiobutton', ...
    'Callback', @radiobending, ...
    'String',   'Clamped␣-␣Clamped', ...
    'Value',    0);
data.bendingBChandle(3) = uicontrol(...
    'Parent', supportcolumn1, ...
    'Style', 'radiobutton', ...
    'Callback', @radiobending, ...
    'String',   'Clamped␣-␣Hinged', ...
    'Value',    0);
data.bendingBChandle(4) = uicontrol(...
    'Parent', supportcolumn2, ...
    'Style', 'radiobutton', ...
    'Callback', @radiobending, ...
    'String',   'Clamped␣-␣Free', ...
    'Value',    0);
data.bendingBChandle(5) = uicontrol(...
    'Parent', supportcolumn2, ...
    'Style', 'radiobutton', ...
    'Callback', @radiobending, ...
    'String',   'Free␣-␣Free', ...
    'Value',    0);
data.bendingBChandle(6) = uicontrol(...
    'Parent', supportcolumn2, ...
    'Style', 'radiobutton', ...
    'Callback', @radiobending, ...
    'String',   'Clamped␣-␣Guided', ...
    'Value',    0);

data.torsionBChandle(1) = uicontrol(...
    'Parent', supportcolumn3, ...
    'Style', 'radiobutton', ...
    'Callback', @radiotorsion, ...
    'String',   'Fixed␣-␣Fixed', ...
    'Value',    1);
data.torsionBChandle(2) = uicontrol(...
    'Parent', supportcolumn3, ...
    'Style', 'radiobutton', ...
    'Callback', @radiotorsion, ...
    'String',   'Fixed␣-␣Free', ...
    'Value',    0);
data.torsionBChandle(3) = uicontrol(...
    'Parent', supportcolumn3, ...
    'Style', 'radiobutton', ...
    'Callback', @radiotorsion, ...
    'String',   'Free␣-␣Free', ...
    'Value',    0);

%-----------------------------------------------------------------------%
%% Input panel geometric properties tab
geometrictab = uiextras.HBox( ...
    'Parent', inputpaneltabs, ...
    'Padding',p, ...
    'Spacing', 25);

% Distribute into columns:
```

```
geometriccolumn1 = uiextras.VBox( ...
    ’Parent’, geometrictab, ...
    ’Spacing’, s);
geometriccolumn2 = uiextras.VBox( ...
    ’Parent’, geometrictab, ...
    ’Spacing’, s);

set(geometrictab, ’Sizes’, [300 300] )
%% Distance between shear center and elastic center:
shearcenterhbox = uiextras.HBox( ...
    ’Parent’, geometriccolumn1, ...
    ’Spacing’, s);
uicontrol( ...
    ’Parent’, shearcenterhbox, ...
    ’Style’,’text’, ...
    ’HorizontalAlignment’,’left’, ...
    ’String’,’c␣(distance␣to␣shear␣center)␣[m]:’);
uicontrol( ...
    ’Parent’, shearcenterhbox, ...
    ’Style’,’edit’, ...
    ’backgroundcol’, [1 1 1], ...
    ’Callback’,@callbackinput, ...
    ’Tag’,’c’);
set(shearcenterhbox, ’Sizes’, [150 100] )
set(geometriccolumn1, ’Sizes’, [efh] )
%% Create an input for beam length:
beamlengthhbox = uiextras.HBox( ...
    ’Parent’, geometriccolumn2, ...
    ’Spacing’, s);
uicontrol( ...
    ’Parent’, beamlengthhbox, ...
    ’Style’,’text’, ...
    ’HorizontalAlignment’,’left’, ...
    ’String’,’Beam␣Length␣[m]:’);
uicontrol( ...
    ’Parent’, beamlengthhbox, ...
    ’Style’,’edit’, ...
    ’backgroundcol’, [1 1 1], ...
    ’Callback’,@callbackinput, ...
    ’Tag’,’beamlength’);
set(beamlengthhbox, ’Sizes’, [150 100] )
%% Create an input for cross section area:
areahbox = uiextras.HBox( ...
    ’Parent’, geometriccolumn2, ...
    ’Spacing’, s);
uicontrol( ...
    ’Parent’, areahbox, ...
    ’Style’,’text’, ...
    ’HorizontalAlignment’,’left’, ...
    ’String’,’Cross␣Section␣Area␣[m^2]:’);
uicontrol( ...
    ’Parent’, areahbox, ...
    ’Style’,’edit’, ...
    ’backgroundcol’, [1 1 1], ...
    ’Callback’,@callbackinput, ...
    ’Tag’,’area’);
set(areahbox, ’Sizes’, [150 100] )
%% Create an input for Iy:
Iyhbox = uiextras.HBox( ...
    ’Parent’, geometriccolumn2, ...
    ’Spacing’, s);
uicontrol( ...
    ’Parent’, Iyhbox, ...
    ’Style’,’text’, ...
    ’HorizontalAlignment’,’left’, ...
    ’String’,’Second␣moment␣of␣area␣Iy␣[m^4]:’);
uicontrol( ...
    ’Parent’, Iyhbox, ...
```

```
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','Iy');
set(Iyhbox, 'Sizes', [150 100] )
%% Create an input for Iz:
Izhbox = uiextras.HBox( ...
    'Parent', geometriccolumn2, ...
    'Spacing', s);
uicontrol( ...
    'Parent', Izhbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Second␣moment␣of␣area␣Iz␣[m^4]:');
uicontrol( ...
    'Parent', Izhbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','Iz');
set(Izhbox, 'Sizes', [150 100] )
set(geometriccolumn2, 'Sizes', [efh efh efh efh] )

%----------------------------------------------------------------------%
%% Input panel material properties tab
materialtab = uiextras.HBox( ...
    'Parent', inputpaneltabs, ...
    'Padding',p);
% Create a vertical box to hold input fields
materialtabvbox = uiextras.VBox( ...
    'Parent', materialtab, ...
    'Spacing',s);
set(materialtab, 'Sizes', [300] )
%% Density:
densityhbox = uiextras.HBox( ...
    'Parent', materialtabvbox, ...
    'Spacing', s);
uicontrol( ...
    'Parent', densityhbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Density␣[kg/m^3]:');
uicontrol( ...
    'Parent', densityhbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','density');
set(densityhbox, 'Sizes', [150 100] )
%% Elasticity module:
Emodulehbox = uiextras.HBox( ...
    'Parent', materialtabvbox, ...
    'Spacing', s);
uicontrol( ...
    'Parent', Emodulehbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Elasticity␣Modulus␣[GPa]:');
uicontrol( ...
    'Parent', Emodulehbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','elasticitymodule');
set(Emodulehbox, 'Sizes', [150 100] )
%% Shear modulus:
Shearmodulushbox = uiextras.HBox( ...
    'Parent', materialtabvbox, ...
```

```
    'Spacing', s);
uicontrol( ...
    'Parent', Shearmodulushbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Shear␣Modulus␣[GPa]:');
uicontrol( ...
    'Parent', Shearmodulushbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','shearmodulus');
set(Shearmodulushbox, 'Sizes', [150 100] )
%% Torsion stiffness:
torsionstiffnesshbox = uiextras.HBox( ...
    'Parent', materialtabvbox, ...
    'Spacing', s);
uicontrol( ...
    'Parent', torsionstiffnesshbox, ...
    'Style','text', ...
    'HorizontalAlignment','left', ...
    'String','Torsion␣stiffness␣[m^4]:');
uicontrol( ...
    'Parent', torsionstiffnesshbox, ...
    'Style','edit', ...
    'backgroundcol', [1 1 1], ...
    'Callback',@callbackinput, ...
    'Tag','torsionstiffness');
set(torsionstiffnesshbox, 'Sizes', [150 100] )

set(materialtabvbox, 'Sizes', [efh efh efh efh] )

%-----------------------------------------------------------------------%
%% Tab names for input panel
inputpaneltabs.TabNames = {'Main', 'Animation␣controls', 'Supports', 'Geometric␣Properties', 'Material␣Properties' };
inputpaneltabs.SelectedChild = 1; % Open the program with the first tab active

%-----------------------------------------------------------------------%
%% About tab:
uicontrol( ...
    'Style', 'text', ...
    'String', sprintf('Author:␣Asger␣Juul␣Brunshøj,␣student␣of␣Architectural␣Engineering␣at␣DTU.\nAdvisor:␣Jan␣Becker␣Høgsberg.\n\nThis␣software␣was␣written␣as␣part␣of␣a␣bachelor''s␣project␣at␣DTU,␣the␣Technical␣University␣of␣Denmark,␣for␣the␣Department␣of␣Civil␣Engineering.\n\nThe␣report␣handed␣in␣together␣with␣this␣software␣serves␣as␣documentation.\n\nJune␣2014'), ...
    'Parent', tabs );
tabs.TabNames = {'Main', 'About'};
tabs.SelectedChild = 1; % Open the program with the first tab active

%-----------------------------------------------------------------------%
%% Save handles for the radiobuttons
% This stores handles to the radiobuttons into guidata. As the radiobuttons
% are handled with custom callback functions, they can't be accessed
% automatically through 'guihandles' like all the input fields can.
% Therefore, they are stored in guidata from where they will be accessed by
% the functions that need them.
guidata(window,data)

%-----------------------------------------------------------------------%
%% Set default values for input fields
% This calls defaults.m which sets values for the input fields.
% window is the handle of the GUI window.%
defaults(window)

%-----------------------------------------------------------------------%
end % ends the main function
%-----------------------------------------------------------------------%
```

```
%-----------------------------------------------------------------------%
%% Callback functions for radiobuttons:
% Below are functions that make the radio buttons function as radio buttons
% by only allowing one to be active at a time, so that when one is clicked,
% the others are set to off.
function radiosolver(hObject, EventData)
handles = guihandles(hObject);
data = guidata(hObject);
otherRadio = data.solverhandle(data.solverhandle ~= hObject);
set(otherRadio, 'Value', 0);
set(hObject, 'Value', 1);
if get(data.solverhandle(1),'Value') == 1 % enable the input field when radiobutton is switched
    set(handles.modeshape,'Enable','On');
    set(handles.initialw,'Enable','Off');
    set(handles.initialphi,'Enable','Off');
    set(handles.initialwdot,'Enable','Off');
    set(handles.initialphidot,'Enable','Off');
    set(handles.load,'Enable','Off');
elseif get(data.solverhandle(2),'Value') == 1
    set(handles.modeshape,'Enable','Off');
    set(handles.initialw,'Enable','On');
    set(handles.initialphi,'Enable','On');
    set(handles.initialwdot,'Enable','On');
    set(handles.initialphidot,'Enable','On');
    set(handles.load,'Enable','Off');
else
    set(handles.modeshape,'Enable','Off');
    set(handles.initialw,'Enable','Off');
    set(handles.initialphi,'Enable','Off');
    set(handles.initialwdot,'Enable','Off');
    set(handles.initialphidot,'Enable','Off');
    set(handles.load,'Enable','On');
end
% Have to call callbackinput here explicitly because the radiobuttons are
% set up with these custom callback functions. See description under
% callbackinput function to see what this does:
callbackinput(hObject,EventData);
end

function radiobending(hObject, EventData)
data = guidata(hObject);
otherRadio = data.bendingBChandle(data.bendingBChandle ~= hObject);
set(otherRadio, 'Value', 0);
set(hObject, 'Value', 1);
callbackinput(hObject,EventData);
end
function radiotorsion(hObject, EventData)
data = guidata(hObject);
otherRadio = data.torsionBChandle(data.torsionBChandle ~= hObject);
set(otherRadio, 'Value', 0);
set(hObject, 'Value', 1);
callbackinput(hObject,EventData);
end

%-----------------------------------------------------------------------%
%% 'Compute' pushbutton callback function:
% The compute button does mainly three things. It calls
% collectinput.m to collect input from input fields. It calls solver.m
% which computes the solution from the user input, and lastly it calls
% plotting.m which pre-renders the plots, making them ready for playback by
% the 'Playback' pushbutton.
function compute(hObject,EventData)
handles = guihandles(hObject);
set(handles.playbackbutton, 'Enable', 'Off' ) % disable buttons while preparing new animation
```

```
% set(handles.computebutton, 'Enable', 'Off' ) % not a good idea, since if the user messes up the input and the program stops, the user will have to restart the whole GUI and lose his input. If the compute button stays accessible, the user can fix the input and continue to use the program without error.
pause(0.001) % for some reason the above needs a small delay for the GUI to visually update properly
collectinput(hObject); % collect and store input from the GUI
solver(hObject); % compute solution
plotting(hObject); % pre-render frames
set(handles.playbackbutton, 'Enable', 'On' )
% set(handles.computebutton, 'Enable', 'On' )
end

%-----------------------------------------------------------------------%
%% 'Playback' pushbutton callback function:
% This calls playback.m, which animates the plot figures.
function playbackanimation(hObject,EventData) % Callback for the playback button
handles = guihandles(hObject);
set(handles.playbackbutton, 'Enable', 'Off' ) % disable buttons during playback
set(handles.computebutton, 'Enable', 'Off' ) % disable buttons during playback
pause(0.001) % for some reason the above needs a small delay for the GUI to visually update properly
playback(hObject); % start the playback
set(handles.playbackbutton, 'Enable', 'On' ) % reenable buttons
set(handles.computebutton, 'Enable', 'On' )
end

%-----------------------------------------------------------------------%
%% Callback function that gets called everytime an input field changes value:
% This callback function is used to inform the user that input has changed
% since the last computation. It does two things: It changes the message
% and it changes the value of a flag 'data.inputchanged', which lets other
% functions like playback.m determine what message to display when it is
% done showing the animation
function callbackinput(hObject,EventData)
notify(hObject,sprintf('\nPress␣Compute␣button␣to␣use␣new␣input.'),'temporary');
data = guidata(hObject);
data.inputchanged = 1; % flag that input has changed. This is used by playback.m, to determine what the console should say after playback.
guidata(hObject, data); % update guidata
end
```

B.3 *defaults.m*

```
function defaults(hObject)
% This sets the default values when the GUI is launched.

% Get handles to the gui input fields:
handles = guihandles(hObject);

set(handles.N, 'String','8');
set(handles.beamlength, 'String','40');
set(handles.density, 'String','7000');
set(handles.area, 'String','0.38');
set(handles.c, 'String','0.5883');
set(handles.elasticitymodule, 'String','200');
set(handles.Iy, 'String','0.0360');
set(handles.Iz, 'String','0.2293');
set(handles.shearmodulus, 'String','70');
set(handles.torsionstiffness, 'String','0.0005');
set(handles.fps,'String','20');
set(handles.duration,'String','6');
set(handles.tmax,'String','2');
set(handles.xGyration, 'String','0.5');
set(handles.modeshape,'String','1');
set(handles.load,'String','sin(2*pi*t)');
set(handles.initialw,'String','0.05*sin(pi*x/L)');
set(handles.initialphi,'String','0');
set(handles.initialwdot,'String','0');
set(handles.initialphidot,'String','0');
notify(hObject,'Click␣the␣Compute␣button␣to␣prepare␣an␣animation␣using␣values␣from␣the␣input␣fields.','temporary')
    ;
end
```

B.4 collectinput.m

```
function collectinput(hObject)
% this collects all the input variables and stores them for use by
% solver.m, plotting.m and playback.m

% Get handles to the gui input fields
handles = guihandles(hObject);
% Get GUI data
data = guidata(hObject);

%-----------------------------------------------------------------------%
% collect input values from the gui handles, and store them in data
data.N = eval(get(handles.N, 'String'));
data.L = eval(get(handles.beamlength, 'String'));
data.rho = eval(get(handles.density, 'String'));
data.area = eval(get(handles.area, 'String'));
data.c = eval(get(handles.c, 'String'));
data.E = eval(get(handles.elasticitymodule, 'String'))*10^9; % input as GPa, convert to SI units
data.Iy = eval(get(handles.Iy, 'String'));
data.Iz = eval(get(handles.Iz, 'String'));
data.G = eval(get(handles.shearmodulus, 'String'))*10^9; % input as GPa, convert to SI units
data.K = eval(get(handles.torsionstiffness, 'String'));
data.fps = eval(get(handles.fps,'String'));
data.duration = eval(get(handles.duration,'String'));
data.tmax = eval(get(handles.tmax,'String'));
data.xGyration = eval(get(handles.xGyration, 'String'));
data.modeshape = eval(get(handles.modeshape,'String'));

%------------------%
% allows L to be used by the user in the input fields of initial conditions
% and external load by using the variable "L".
L = data.L;

%------------------%
% The load function
string = get(handles.load,'String');
prefix = '@(x,t)';
InKiloNewtons = eval(strcat(prefix,string));
data.p = @(x,t) 1000*InKiloNewtons(x,t); % convert to newtons

%------------------%
% Initial conditions
string = get(handles.initialw,'String');
prefix = '@(x)';
data.initialw = eval(strcat(prefix,string));

string = get(handles.initialphi,'String');
prefix = '@(x)';
data.initialphi = eval(strcat(prefix,string));

string = get(handles.initialwdot,'String');
prefix = '@(x)';
data.initialwdot = eval(strcat(prefix,string));

string = get(handles.initialphidot,'String');
prefix = '@(x)';
data.initialphidot = eval(strcat(prefix,string));

%------------------%
% The buttons that are active one at a time controlling the support conditions etc.
% (called radio buttons), have to be accessed through guidata instead of guihandles
% because of how they are set up.
% The following will create vectors like [0 1 0] which will be used in solver.m
% to select the correct basis functions for the chosen support conditions
data.bendingBC = [get(data.bendingBChandle(1),'Value');
                  get(data.bendingBChandle(2),'Value');
```

```
                    get(data.bendingBChandle(3),'Value');
                    get(data.bendingBChandle(4),'Value');
                    get(data.bendingBChandle(5),'Value');
                    get(data.bendingBChandle(6),'Value')];
data.torsionBC = [get(data.torsionBChandle(1),'Value');
                    get(data.torsionBChandle(2),'Value');
                    get(data.torsionBChandle(3),'Value')];

%-------------------%
% This is the radio buttons that determines whether to display a single modeshape,
% a harmonic oscillation from initial conditions, or a numerical simulation of a load
data.solver = [get(data.solverhandle(1),'Value');
                    get(data.solverhandle(2),'Value');
                    get(data.solverhandle(3),'Value')];


%--------------------------------------------------------------------------%
% The following values are not direct input values, but are derived from the above
data.Ip = data.Iy + data.Iz; % polar moment of area.
data.nframes = max([round(data.duration * data.fps) 2]); % compute the number of frames in the animations. max()
     with the 2 ensures that there is at least 2 frames. This means a little less care can be taking in solver.m
     in certain places because 1 frame would lead to vectors instead of arrays. Another effect of this is that if
     the user messes up the inputs for animation duration and fps in relation to each other badly enough so
     nframes would be calculated as 0 (like a user who is interested only in the coupling image, not the animation
      and want it to compute as fast as possible), then the whole program crashes quite badly, and the whole thing
      has to be restarted leading to reentering input. This just safeguards against that.
data.tpoints = linspace(0,data.tmax,data.nframes); % compute time values for each frame

% Updata guidata with the new input
guidata(hObject,data);
end
```

B.5 solver.m

```
function solver(hObject)
% This function computes the vibration based on input collected from the GUI
% input fields.

%----------------------------------------------------------------------%
% Get guihandles and guidata collected from input fields:
handles = guihandles(hObject);
data = guidata(hObject);
% For convenience of notation since these are used repeatedly:
N = data.N;
L = data.L;
tpoints = data.tpoints;
c = data.c;
rho = data.rho;
Ip = data.Ip;
area = data.area;

%----------------------------------------------------------------------%
% The samples over the x axes with values from 0 to L:
xpoints = 0:0.002*L:L; % provides about 500 points, which is about 1 point per horizontal pixel in the animation at default window size.

%----------------------------------------------------------------------%
%% Bending Basis Functions
if data.bendingBC(1) == 1
    [W, alpha] = hinged_hinged_BENDING(N,xpoints,L);
elseif data.bendingBC(2) == 1
    [W, alpha] = clamped_clamped_BENDING(N,xpoints,L);
elseif data.bendingBC(3) == 1
    [W, alpha] = clamped_hinged_BENDING(N,xpoints,L);
elseif data.bendingBC(4) == 1
    [W, alpha] = clamped_free_BENDING(N,xpoints,L);
elseif data.bendingBC(5) == 1
    [W, alpha] = free_free_BENDING(N,xpoints,L);
else
    [W, alpha] = clamped_guided_BENDING(N,xpoints,L);
end

% Reverse
if get(handles.reversebending, 'Value') == 1
    W = fliplr(W);
end

%% Torsion Basis Functions
if data.torsionBC(1) == 1
    [Phi, beta] = fixed_fixed_TORSION(N,xpoints,L);
elseif data.torsionBC(2) == 1
    [Phi, beta] = fixed_free_TORSION(N,xpoints,L);
else
    [Phi, beta] = free_free_TORSION(N,xpoints,L);
end

% Reverse
if get(handles.reversetorsion, 'Value') == 1
    Phi = fliplr(Phi);
end

%----------------------------------------------------------------------%
%% Coupling integrals
Psi = zeros(N,N); % initialize
% A row in Psi corresponds to the index on W, while a column
% corresponds to the index on Phi.
for i=1:N
    for j=1:N
        Psi(i,j) = c*rho*area*trapz(xpoints,W(i,:).*Phi(j,:));
```

```
    end
end

%----------------------------------------------------------------------%
%% Mass Matrix
M = [rho*area*diag(ones(N,1))       -Psi;
     -Psi' (c^2*rho*area+rho*Ip)*diag(ones(N,1))];

%----------------------------------------------------------------------%
%% Stiffness Matrix
K = [data.E*data.Iz*diag(alpha).^4*diag(ones(N,1)) zeros(N);
     zeros(N)                                      data.G*data.K*diag(beta).^2*diag(ones(N,1))];

%----------------------------------------------------------------------%
%% Solve the eigenproblem
[natfreq,eigenvectors] = EigenProblemSolver(M,K,hObject);
printnatfreq(hObject,N,natfreq) % show the natural frequencies to the user
data.eigenvectors = eigenvectors; % makes it available to plotting.m

%----------------------------------------------------------------------%
%% Choose a solver subfunction, and compute all the values for w and phi:
if data.solver(1) == 1 % 'Individual Mode Shape' chosen
    message = sprintf('\nSolving␣for␣the␣%d.␣Mode␣Shape...',data.modeshape);
    notify(hObject,message,'append');
    [w,phi] = singlemodeshape(N,natfreq,eigenvectors,W,Phi,tpoints,handles,data,hObject);
elseif data.solver(2) == 1 % 'Harmonic Oscillation With Initial Conditions' chosen
    notify(hObject,sprintf('\nSolving␣with␣initial␣conditions...'),'append');
    [w,phi] = naturalresponse(N,natfreq,eigenvectors,W,Phi,xpoints,tpoints,handles,data,hObject);
else % 'Specified Load' chosen
    notify(hObject,sprintf('\nSolving␣with␣numerical␣solver...'),'append');
    [w,phi] = forcedresponse(L,N,M,K,W,Phi,xpoints,tpoints,data);
end

%----------------------------------------------------------------------%
%% Warn the user if the absolute torsion is above 0.1 radians,
% A torsion above 0.1 radians will break with the assumption of linearity.
if max(abs(phi(:))) > 0.101
    notify(hObject,sprintf('\nWARNING:␣Torsion␣is␣over␣0.1␣radians,␣which␣violates␣the␣assumption␣of␣linearity!'),
         'append');
end

%----------------------------------------------------------------------%
%% Update guidata to make the results from this file (saved into "data") available from other subfiles like
     playback.m
data.xpoints = xpoints;
data.w = w;
data.phi = phi;
guidata(hObject, data); % updates guidata. hObject is just a handle to the instance of the program that is calling
      this file.

%----------------------------------------------------------------------%
%% THIS ENDS THE MAIN FUNCTION
end


%----------------------------------------------------------------------%
%% Functions for bending:
% The following defines nested functions for different boundary conditions.
% Only one of these is called.
% basisfunction is an array. It is the basis functions for bending.
% The rows corresponds to values of n, while the
% columns corresponds to values of x.
% In the following, ndgrid is used to create arrays out of vectors,
% which allows basisfunction (which is really a function of two variables,
% W_n(x) or Phi_n(x), to be constructed as an array.
function [basisfunction, spatialfreq] = hinged_hinged_BENDING(N,xpoints,L)
roots = (1:N)*pi;
[rG, xG] = ndgrid(roots,xpoints); % This order will mean that points are accessed as W(n,x) or Phi(n,x)
```

```
basisfunctionRaw = sin(rG.*xG/L);
NormalizationFactor = sqrt(2/L);
basisfunction = basisfunctionRaw*NormalizationFactor;
spatialfreq = roots/L;
end

function [basisfunction, spatialfreq] = clamped_clamped_BENDING(N,xpoints,L)
precomputed = [4.7300 7.8532 10.9956 14.1372];
if N<5
    roots = precomputed(1:N);
else
    roots = [precomputed (2*(5:N)+1)*pi/2];
end
[rG, xG] = ndgrid(roots,xpoints);
basisfunctionRaw = cosh(rG.*xG/L)-cos(rG.*xG/L)-(cosh(rG)-cos(rG))./(sinh(rG)-sin(rG)).*(sinh(rG.*xG/L)-sin(rG.*xG/L));
NormalizationFactor = 1./sqrt(trapz(xpoints,(basisfunctionRaw.^2)')); % this is a vector of constants D_n. These are computed here so that the integral of W^2 from 0 to L is 1.
basisfunction = diag(NormalizationFactor)*basisfunctionRaw; % the normalization factor has to be multiplied onto the rows of basisfunctionRaw.
spatialfreq = roots/L;
end

function [basisfunction, spatialfreq] = clamped_hinged_BENDING(N,xpoints,L)
precomputed = [3.9266 7.0686 10.2102 13.3518];
if N<5
    roots = precomputed(1:N);
else
    roots = [precomputed (4*(5:N)+1)*pi/4];
end
[rG, xG] = ndgrid(roots,xpoints);
basisfunctionRaw = cosh(rG.*xG/L)-cos(rG.*xG/L)-(cosh(rG)-cos(rG))./(sinh(rG)-sin(rG)).*(sinh(rG.*xG/L)-sin(rG.*xG/L));
NormalizationFactor = 1./sqrt(trapz(xpoints,(basisfunctionRaw.^2)'));
basisfunction = diag(NormalizationFactor)*basisfunctionRaw;
spatialfreq = roots/L;
end

function [basisfunction, spatialfreq] = clamped_free_BENDING(N,xpoints,L)
precomputed = [1.8751 4.6941 7.8548 10.9955];
if N<5
    roots = precomputed(1:N);
else
    roots = [precomputed (2*(5:N)-1)*pi/2];
end
[rG, xG] = ndgrid(roots,xpoints);
basisfunctionRaw = cosh(rG.*xG/L)-cos(rG.*xG/L)-(cosh(rG)+cos(rG))./(sinh(rG)+sin(rG)).*(sinh(rG.*xG/L)-sin(rG.*xG/L));
NormalizationFactor = 1./sqrt(trapz(xpoints,(basisfunctionRaw.^2)'));
basisfunction = diag(NormalizationFactor)*basisfunctionRaw;
spatialfreq = roots/L;
end

function [basisfunction, spatialfreq] = free_free_BENDING(N,xpoints,L)
precomputed = [4.7300 7.8532 10.9956 14.1372];
if N<5
    roots = precomputed(1:N);
else
    roots = [precomputed (2*(5:N)+1)*pi/2];
end
[rG, xG] = ndgrid(roots,xpoints);
basisfunctionRaw = cosh(rG.*xG/L)+cos(rG.*xG/L)-(cosh(rG)-cos(rG))./(sinh(rG)-sin(rG)).*(sinh(rG.*xG/L)+sin(rG.*xG/L));
NormalizationFactor = 1./sqrt(trapz(xpoints,(basisfunctionRaw.^2)'));
basisfunction = diag(NormalizationFactor)*basisfunctionRaw;
spatialfreq = roots/L;
end
```

```
function [basisfunction, spatialfreq] = clamped_guided_BENDING(N,xpoints,L)
precomputed = [2.3650 5.4978 8.6394 11.7810];
if N<5
    roots = precomputed(1:N);
else
    roots = [precomputed (4*(5:N)-1)*pi/4];
end
[rG, xG] = ndgrid(roots,xpoints);
basisfunctionRaw = cosh(rG.*xG/L)-cos(rG.*xG/L)-(sinh(rG)+sin(rG))./(cosh(rG)-cos(rG)).*(sinh(rG.*xG/L)-sin(rG.*xG/L));
NormalizationFactor = 1./sqrt(trapz(xpoints,(basisfunctionRaw.^2)'));
basisfunction = diag(NormalizationFactor)*basisfunctionRaw;
spatialfreq = roots/L;
end

%----------------------------------------------------------------------%
%% Functions for torsion:
function [basisfunction, spatialfreq] = fixed_fixed_TORSION(N,xpoints,L)
% This has the same solution as hinged_hinged bending:
[basisfunction, spatialfreq] = hinged_hinged_BENDING(N,xpoints,L);
end

function [basisfunction, spatialfreq] = fixed_free_TORSION(N,xpoints,L)
roots = (2*(1:N)-1)*pi/2;
[rG, xG] = ndgrid(roots,xpoints);
basisfunctionRaw = sin(rG.*xG/L);
NormalizationFactor = 1./sqrt(trapz(xpoints,(basisfunctionRaw.^2)'));
basisfunction = diag(NormalizationFactor)*basisfunctionRaw;
spatialfreq = roots/L;
end

function [basisfunction, spatialfreq] = free_free_TORSION(N,xpoints,L)
roots = (1:N)*pi;
[rG, xG] = ndgrid(roots,xpoints);
basisfunctionRaw = cos(rG.*xG/L);
NormalizationFactor = 1./sqrt(trapz(xpoints,(basisfunctionRaw.^2)'));
basisfunction = diag(NormalizationFactor)*basisfunctionRaw;
spatialfreq = roots/L;
end

%----------------------------------------------------------------------%
% The following defines four functions.
% Three for the different problems
% of showing a single modeshape, starting the system with specified initial
% conditions, and applying a load to the system.
% And a fourth EigenProblemSolver, which is called before the others.
function [natfreq,eigenvectors] = EigenProblemSolver(M,K,hObject)
[eigenvectors,eigenvalues] = eig(K,M);

% create vector with naturalfrequencies:
natfreq = diag(sqrt(real(eigenvalues))); % the contents of 'eigenvalues' are real, but are sometimes represented as e.g.: 1.0000 + 0.0000i, so real() is just removing the 0.0000i

%-----------------------%
% This warns the user if a bug is detected, see "known bugs" in the report:
if min(min(eigenvalues)) < 0 % negative eigenvalues seem to follow when eig() produces weird output
    notify(hObject,sprintf('\n\nWARNING:␣eig()␣HAS_PRODUCED␣WEIRD␣OUTPUT.␣SEE␣"KNOWN␣BUGS"␣IN␣REPORT.␣Likely␣this␣happened␣because␣the␣eigenvalues␣computed␣by␣eig()␣are␣extremely␣small␣or␣extremely␣big.␣One␣reason␣why␣this␣may␣have␣happened␣is␣that␣N␣is␣set␣to␣a␣large␣number,␣so␣that␣many␣natural␣frequncies␣are␣computed,␣where␣the␣last␣of␣them␣become␣too␣big␣to␣handle.␣You␣may␣try␣again␣with␣different␣input.\n'),'reset');
    error('WARNING:␣eig()␣HAS␣PRODUCED␣WEIRD␣OUTPUT.␣SEE␣"KNOWN␣BUGS"␣IN␣REPORT.␣Likely␣this␣happened␣because␣the␣eigenvalues␣computed␣by␣eig()␣are␣extremely␣small␣or␣extremely␣big.')
end
end

%----------------------------------------------------------------------%
function [w,phi] = singlemodeshape(N,natfreq,eigenvectors,W,Phi,tpoints,handles,data,hObject)
% The report handed in together with this software has a deeper explanation
```

```
% of how this works, so please use that.

%--------------------%
% The sum turned into a vector with the value of the sum at any point x (the sum is a function of only x):
sumcontent = zeros(size(W)); % initialize
for i = 1:N
    sumcontent(i,:) = W(i,:) * eigenvectors(i,data.modeshape);
end

if ~isrow(sumcontent) % sumcontent is an array
    sumterm = sum(sumcontent);
else % sumcontent is a row vector, this happens in the special case that N is set to 1, and in this case we definitely don't want to use sum() on it, because it would sum by the row, not by the column as it does when given an array.
    sumterm = sumcontent;
end

[sumtermGRID,tpointsGRID] = ndgrid(sumterm,tpoints);
w = cos(natfreq(data.modeshape)*tpointsGRID) .* sumtermGRID;

%--------------------%
sumcontent = zeros(size(Phi)); % initialize
for i = 1:N
    sumcontent(i,:) = Phi(i,:) * eigenvectors(N+i,data.modeshape);
end

if ~isrow(sumcontent) % sumcontent is an array
    sumterm = sum(sumcontent);
else % sumcontent is a row vector, this happens when N is set to 1, and in this case we definitely don't want to use sum() on it, because it would sum by the row, not by the column as it does when given an array.
    sumterm = sumcontent;
end

[sumtermGRID,tpointsGRID] = ndgrid(sumterm,tpoints);
phi = cos(natfreq(data.modeshape)*tpointsGRID) .* sumtermGRID;

%--------------------%
% scale with regards to rotation
maxTorsionAmplitude = max(abs(phi(:,1))); % t = 0
maxBendingAmplitude = max(abs(w(:,1))); % t = 0
if maxBendingAmplitude / maxTorsionAmplitude < 50 % if true, scale by rotation. This is a cap for the amount of bending we will see. This is relevant fx if the distance between the shear center and center of mass is given a small value.
    phi = phi / maxTorsionAmplitude * 0.1;
    w = w / maxTorsionAmplitude * 0.1;
else % Rotation is too small to scale by rotation, and we will just scale by displacement instead.
    phi = phi / maxBendingAmplitude * 5; % the last factor here maximizes the amplitudes, while securing that rotation stays below 0.1 (because the ratio above is 50)
    w = w / maxBendingAmplitude * 5;
end
end

%----------------------------------------------------------------------%
function [w,phi] = naturalresponse(N,natfreq,eigenvectors,W,Phi,xpoints,tpoints,handles,data,hObject)
% The report handed in together with this software has a deeper explanation
% of how this works, so please use that.

% First arrays Ww, Wwdot, Phiphi and Phiphidot are computed, which,
% taking Ww as an example, are W_k(x)*w(x,0). These are the contents of the
% integrals on the right hand side of the matrix equation defining the
% constants A_k and B_k.
initialw = data.initialw(xpoints); % compute points from function
Ww = zeros(size(W)); % initialize
for i = 1:N
    Ww(i,:) = W(i,:) .* initialw;
end
initialwdot = data.initialwdot(xpoints); % compute points from function
Wwdot = zeros(size(W)); % initialize
```

```
for i = 1:N
    Wwdot(i,:) = W(i,:) .* initialwdot;
end
initialphi = data.initialphi(xpoints); % compute points from function
Phiphi = zeros(size(Phi)); % initialize
for i = 1:N
    Phiphi(i,:) = Phi(i,:) .* initialphi;
end
initialphidot = data.initialphidot(xpoints); % compute points from function
Phiphidot = zeros(size(Phi)); % initialize
for i = 1:N
    Phiphidot(i,:) = Phi(i,:) .* initialphidot;
end
% B:
rhs = [trapz(xpoints,Ww') trapz(xpoints,Phiphi')]';
B = eigenvectors\rhs;
% A:
rhs = [trapz(xpoints,Wwdot') trapz(xpoints,Phiphidot')]';
A = (eigenvectors*diag(natfreq(:)))\rhs;

w = zeros(length(xpoints),length(tpoints)); % initialize
phi = zeros(length(xpoints),length(tpoints));
for n = 1:N
    sumterm = 0;
    for k = 1:2*N
        sumterm = sumterm + (A(k)*sin(natfreq(k)*tpoints)+B(k)*cos(natfreq(k)*tpoints)) * eigenvectors(n,k);
    end
    [WGRID,sumtermGRID] = ndgrid(W(n,:),sumterm);
    w = w + WGRID .* sumtermGRID;

    sumterm = 0;
    for k = 1:2*N
        sumterm = sumterm + (A(k)*sin(natfreq(k)*tpoints)+B(k)*cos(natfreq(k)*tpoints)) * eigenvectors(N+n,k);
    end
    [PhiGRID,sumtermGRID] = ndgrid(Phi(n,:),sumterm);
    phi = phi + PhiGRID .* sumtermGRID;
end

end

%-----------------------------------------------------------------------%
function [w,phi] = forcedresponse(L,N,M,K,W,Phi,xpoints,tpoints,data)
% The report handed in together with this software has a deeper explanation
% of how this works, so please use that.

H = [zeros(2*N) eye(2*N);
    -M\K zeros(2*N)];
    function odeRHS = RHS(t,q)
        pvec = data.p(xpoints,t);
        % note in the above line, that if the user did not type x in the formula
        % for p (like a constant evenly distributed load of 100 N/m which would
        % be input as just "100" with no x or t in it), then the above evaluates
        % to a scalar, not a vector. This isn't a problem in the following, but
        % if the code is altered, pvec should be expanded into a vector when it
        % evaluates to a scalar.

        f = zeros(2*N,1); % initialize
        for n = 1:N
            f(n) = -trapz(xpoints,W(n,:).*pvec);
        end
        for n = 1:N
            f(N+n) = trapz(xpoints,Phi(n,:).*pvec*data.c);
        end
        b = [zeros(2*N,1); M\f];
        odeRHS = H*q+b;
    end
initial = zeros(4*N,1); % % This means that the animation is started from rest. The first N rows represent time
     functions for bending, r_1(0), r_2(0), ..., while the next N rows are s_1(0), s_2(0),... Then comes N initial
```

```
        derivatives for bending, r'(0), and lastly N initial derivatives for torsion s'(0).

[T,Q] = ode45(@RHS,tpoints,initial); % The first quarter of columns of this solution is for r(t), the next quarter
        for s(t), and the last half is the first derivative (irrelevant)

% split into r(t) and s(t)
r = Q(:,1:N); % the rows are time, and the columns are r_1(t), r_2(t) and so on.
s = Q(:,N+1:2*N);

w = zeros(length(xpoints),length(tpoints));
phi = zeros(length(xpoints),length(tpoints));
for i = 1:N
    [WGRID,rGRID] = ndgrid(W(i,:),r(:,i));
    w = w + WGRID.*rGRID;

    [PhiGRID,sGRID] = ndgrid(Phi(i,:),s(:,i));
    phi = phi + PhiGRID.*sGRID;
end

end

function printnatfreq(hObject,N,natfreq)
% Shows the natural frequencies to the user
notify(hObject,sprintf('N = %d.\n\nThe first natural frequencies are:\n[in cycles per second] ',N),'reset')
% convert from radians per second to cycles per second:
natfreq = natfreq/(2*pi);
% round to two decimals:
natfreq = roundn(natfreq,-2);
if N<8
    notify(hObject,natfreq,'append');
else
    notify(hObject,natfreq(1:14),'append');
end

end
```

B.6 *plotting.m*

```
function plotting (hObject)
% This function renders the animations and static graphics.

handles = guihandles(hObject);
data = guidata(hObject);
notify(hObject,sprintf('\nPre-rendering␣animation...'),'temporary');

%----------------------------------------------------------------------%
%% Clear the axes:
cla(data.animationright)
cla(data.animationleft)
cla(data.staticaxesbending)
cla(data.staticaxestorsion)
cla(data.staticaxescoupling)

%----------------------------------------------------------------------%
%% Static images:
% To ensure that the scale is the same in the two static images, first
% compute maximum values:
absmaxB = max(abs(data.w(:)));
absmaxT = max(abs(data.phi(:)));
absmaxBT = max([absmaxB absmaxT]);
% Update the axes:
imagesc(flipud(data.w'), 'Parent',data.staticaxesbending,[-absmaxBT absmaxBT]);
set(data.staticaxesbending,'XTickLabel','','YTickLabel','') % removes tick marks, since these would take a little
     work to get right. At present, they would be 0 to length(xpoints) on the first axes, and 0 to length(tpoints)
      on the second axes. You would want 0 to L on the first axes and 0 to tmax on the second.
imagesc(flipud(data.phi'), 'Parent',data.staticaxestorsion,[-absmaxBT absmaxBT]);
set(data.staticaxestorsion,'XTickLabel','','YTickLabel','')
% Update the image in the "Visualize Coupling" output tab:
imagesc(abs(data.eigenvectors), 'Parent',data.staticaxescoupling);

%----------------------------------------------------------------------%
%% The animations:

% 'xGyrIndex' is the index of data.xpoints, corresponding to the x coordinate
% that the left animation will show gyration radius displacement and rotation
% for. So for example, if data.xpoints has 500 points and the user
% specifyes 0.25 for data.xGyration, xGyrIndex will be 0.25*500 = 125. It
% is rounded because it is an index:
xGyrIndex = round(length(data.xpoints)*data.xGyration);
% if the user specified either 0 or 1, trying to get the end points, the above
% may be unsuccessful because of rounding. In that case, the following
% straightens it out:
if xGyrIndex < 1
    xGyrIndex = 1;
elseif xGyrIndex > length(data.xpoints)
    xGyrIndex = length(data.xpoints);
end
% the gyration radii are computed from the Area, Iy and Iz
ry = sqrt(data.Iy/data.area);
rz = sqrt(data.Iz/data.area);

% Axes must have hold on or layerGyration(1) will become invalid after
% layerGyration(2) is created and so on. This is a technical point,
% needed because of how the animation is constructed in layers.
hold(data.animationleft,'on');
hold(data.animationright,'on');

%% The Left Plot (Gyration axes):

% The left animation must have equal axis spacing. Otherwise, the two lines
% would not even stay perpendicular:
axis(data.animationleft,'equal');
% During the loop that generates the frames, the following four variables
```

```
% keep track of how far from (0,0) the content of the animation get. This
% is used in scaling the animation frame. First they are reset to 0:
xmin = 0;
xmax = 0;
ymin = 0;
ymax = 0;

% The following loop prepares the perpendicular lines that make up the left
% plot, by rotating and translating them
for frame=1:data.nframes % loop over all frames
    % Get the twist angle for this frame from the solution computed in solver.m:
    angle = data.phi(xGyrIndex,frame);
    rotationmatrix = [cos(angle) -sin(angle); ...
        sin(angle)  cos(angle)]; % counter-clockwise rotation matrix

    % The horizontal line:
    % (the format [x1 x2; y1 y2] is a line from (x1,y1) to (x2,y2),
    % so the lines are in columns)
    hline = [-ry  ry;
        0   0];
    hlinerot = (rotationmatrix*hline); % the horizontal line, now rotated
    % The vetical line:
    vline = [ 0   0;
        -rz  rz];
    vlinerot = (rotationmatrix*vline); % the vertical line, now rotated

    % When using the above lines with MATLABs plot(), the format has to be
    % a little different. The following changes the format to suit plot()
    hlinerotx = hlinerot(1,:); % all the x coordinates to the horizontal line
    hlineroty = hlinerot(2,:); % all the y coordinates to the horizontal line
    vlinerotx = vlinerot(1,:);
    vlineroty = vlinerot(2,:);

    % The lines are now rotated. The following displaces the lines by
    % correcting the y coordinates. The vertical displacement of the elastic center
    % is w-c*sin(phi), and the horizontal displacement is c*cos(phi).
    % Notice that here the trigonometric functions are used,
    % to correctly display the displacement, whereas the computation assumes
    % small angles leading to linearity, w-c*phi.
    xDisplacement = data.c*(1-cos(data.phi(xGyrIndex,frame)));
    yDisplacement = data.w(xGyrIndex,frame) - data.c*sin(data.phi(xGyrIndex,frame));
    hlinerotx = hlinerotx + xDisplacement;
    vlinerotx = vlinerotx + xDisplacement;
    hlineroty = hlineroty + yDisplacement;
    vlineroty = vlineroty + yDisplacement;

    % and we're ready to render the plots one line at a time
    layerhline(frame) = plot(data.animationleft,hlinerotx,hlineroty,'Color','k','visible','off');
    layervline(frame) = plot(data.animationleft,vlinerotx,vlineroty,'Color','k','visible','off');
    layerCenterofMass(frame) = plot(data.animationleft,xDisplacement,yDisplacement,'Color','b','Marker','*','visible','off');
    layerShearCenter(frame) = plot(data.animationleft,data.c,data.w(xGyrIndex,frame),'Color','red','Marker','*','visible','off');

    %% The axes have to be scaled carefully to ensure that the frame captures
    %% the whole animation. The following keeps track of the most outlying
    %% points during the animation:
    tempxmin = min([hlinerotx vlinerotx data.c]);
    tempxmax = max([hlinerotx vlinerotx data.c]);
    tempymin = min([hlineroty vlineroty data.w(xGyrIndex,frame)]);
    tempymax = max([hlineroty vlineroty data.w(xGyrIndex,frame)]);
    if xmin > tempxmin
        xmin = tempxmin;
    end
    if xmax < tempxmax
        xmax = tempxmax;
    end
    if ymin > tempymin
```

```
        ymin = tempymin;
    end
    if ymax < tempymax
        ymax = tempymax;
    end
end
% Add a little padding to the plotrange:
xpadding = 0.03*(xmax-xmin);
ypadding = 0.03*(ymax-ymin);
% Create two invisible points which are nonetheless included in the plot at all times.
% They are created at the outmost lower left corner and upper right corner of the
% animation, which keeps MATLAB from scaling the plotrange dynamically as the animation
% is played back:
plot(data.animationleft,xmin-xpadding,-max([abs(ymin) ymax])-ypadding,'Color','white');
plot(data.animationleft,xmax+xpadding,max([abs(ymin) ymax])+ypadding,'Color','white');


%% The Right Plot (Twist and Bending Curves):

% The largest y-coordinate of any point in the right animation:
ymaxright = max([absmaxB absmaxT 0.001]);
% Set the limits to the right animation:
axis(data.animationright,[0,data.L,-ymaxright,ymaxright]);
% line on the right plot that shows which x coordinate the left plot is focused on
x = [data.xpoints(xGyrIndex) data.xpoints(xGyrIndex)];
y = [-ymaxright -0.6*ymaxright];
plot(data.animationright,x,y,'b','visible','on');

% Render the frames of the right animations:
for frame=1:data.nframes
    %bending curve
    y = data.w(:,frame);
    layerBendingCurve(frame) = plot(data.animationright,data.xpoints,y,'k','visible','off');

    %twist curve
    y = data.phi(:,frame);
    layerTorsionCurve(frame) = plot(data.animationright,data.xpoints,y,'r','visible','off');
end

% Decorate the animations with labels and legend
legend(data.animationright,'Left␣plot␣x-position','Deflection␣due␣to␣bending,␣w(x,t)','Angle␣of␣twist,␣phi(x,t)');
xlabel(data.animationright,'x␣[m]')
ylabel(data.animationright,'z␣[m]␣/␣Rotation␣[rad]')
xlabel(data.animationleft,'y␣[m]')
ylabel(data.animationleft,'z␣[m]')

% Show the first frame after pre-rendering:
set(layerhline(1),'visible','on');
set(layervline(1),'visible','on');
set(layerBendingCurve(1),'visible','on');
set(layerTorsionCurve(1),'visible','on');
set(layerCenterofMass(1),'visible','on');
set(layerShearCenter(1),'visible','on');

% save the layers that make up the plots, so they can be turned off and on
% by playback.m:
data.layerhline = layerhline;
data.layervline = layervline;
data.layerBendingCurve = layerBendingCurve;
data.layerTorsionCurve = layerTorsionCurve;
data.layerShearCenter = layerShearCenter;
data.layerCenterofMass = layerCenterofMass;

data.inputchanged = 0; % reset the flag because the animation is now up to date (this flag is used when showing messages to the user)

notify(hObject,sprintf('\nReady␣for␣playback!'),'temporary');

```

```
guidata(hObject, data); % Update guidata
end
```

B.7 playback.m

```
function playback (hObject)
% This is a function that animates the plot by turning
% layers on the plot on and off, which admittedly is an odd way of doing it.
% I found it necessary to write this function, because out of the box,
% MATLAB with its build in functions like movie(), cannot animate two plots
% at the same time. This looping over frames allows the animations to be
% rendered 'in parallel'.

handles = guihandles(hObject);
data = guidata(hObject);

for frame=1:data.nframes
    % Turn off previous frame:
    if frame>1 % dont do this on first iteration - will cause index out of bounds
        set(data.layerhline(frame-1),'visible','off');
        set(data.layervline(frame-1),'visible','off');
        set(data.layerBendingCurve(frame-1),'visible','off');
        set(data.layerTorsionCurve(frame-1),'visible','off');
        set(data.layerShearCenter(frame-1),'visible','off');
        set(data.layerCenterofMass(frame-1),'visible','off');
    end
    % Turn on current frame:
    set(data.layerhline(frame),'visible','on');
    set(data.layervline(frame),'visible','on');
    set(data.layerBendingCurve(frame),'visible','on');
    set(data.layerTorsionCurve(frame),'visible','on');
    set(data.layerShearCenter(frame),'visible','on');
    set(data.layerCenterofMass(frame),'visible','on');

    message = sprintf('t␣=␣%.3f␣seconds',data.tpoints(frame));
    set(handles.animationtime,'String',message);

    pause(1/data.fps)

    % if this is the last frame, remove it and show the first frame again.
    % This makes it a little nicer because when the playback is not
    % running, the figures will show the initial conditions.
    if frame == data.nframes(end)
        set(data.layerhline(frame),'visible','off');
        set(data.layervline(frame),'visible','off');
        set(data.layerBendingCurve(frame),'visible','off');
        set(data.layerTorsionCurve(frame),'visible','off');
        set(data.layerShearCenter(frame),'visible','off');
        set(data.layerCenterofMass(frame),'visible','off');
        set(data.layerhline(1),'visible','on');
        set(data.layervline(1),'visible','on');
        set(data.layerBendingCurve(1),'visible','on');
        set(data.layerTorsionCurve(1),'visible','on');
        set(data.layerShearCenter(1),'visible','on');
        set(data.layerCenterofMass(1),'visible','on');
    end
end
set(handles.animationtime,'String','');

end
```

B.8 notify.m

```
function notify(hObject,message,option)
% This function is used to update text based information in the left side of the GUI
% It has three modes:
% 'reset'   Removes all previous text
% 'append'    Appends the message to the previous message.
% 'temporary' Appends to the previous text, but does not save the appended message,
%       so it will disappear the next time this function is called, no matter
%       the option.

handles = guihandles(hObject);

if strcmp(option,'reset') % Previous message is cleared first.
  set(handles.console, 'String', message);
  set(handles.console, 'UserData', {message});
elseif strcmp(option,'append') % Message is appended to previous message.
  newmessage = get(handles.console, 'UserData');
  newmessage{end+1} = message;
  set(handles.console, 'String', newmessage);
  set(handles.console, 'UserData', newmessage);
elseif strcmp(option,'temporary') % Message is appended, but only temporarily. It is not saved into data.
    currentmessage, and therefore disappears the next time notify is called.
  newmessage = get(handles.console, 'UserData');
  newmessage{end+1} = message;
  set(handles.console, 'String', newmessage);
end

pause(0.0001); % for some reason, the GUI does not update if MATLAB is about to do something else, like when
    notify is called from within solver.m. This forces MATLAB to stop for a moment and gives it time to update
    the GUI. It seems to work.

end
```

Bibliography

The program code. URL https://dl.dropboxusercontent.com/u/7180193/BA/BAprogram.zip.

Ole Christensen. *Differentialligninger og uendelige rækker*. 2005. ISBN 87-88-76473-7.

Jan Becker Høgsberg and Steen Krenk. Analysis of moderately thin-walled beam cross-sections by cubic isoparametric elements. *Computers and Structures*, 134:88–101, 2014. ISSN 0045-7949. DOI: 10.1016/j.compstruc.2014.01.002.

Daniel. Inman. *Engineering Vibration*. Pearson Prentice Hall, 2008. ISBN 0132281732, 9780132281737.

Steen Krenk and Jan Becker Høgsberg. *Statics and Mechanics of Structures*. 2013. ISBN 978-94-007-6112-4.

Jon Juel Thomsen. *Vibrations and Stability*. 2003. ISBN 978-3-642-07272-7.