FEM Practices

Jan Tomek

December 5, 2023

Contents

Chapter 1

Coordinate Transformations

Contents

1.1   Constructing Transformation Matrix

1.1.1   Rotation Matrix

From: wolfram

1. 2-D (in $\mathbb{R}^2$):

   Rotating an object to a new coordinate system:
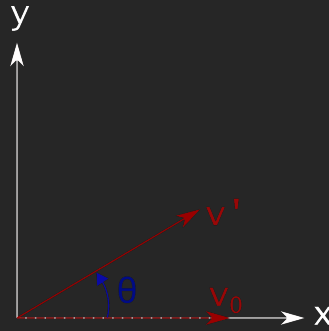
Figure 1.1: Rotating object

In $\mathbb{R}^2$, consider the matrix that rotates a given vector $\mathbf{v}_0$ by a counter-clockwise angle θ in a fixed coordinate system. Then

$$\mathbf{T}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \tag{1.1}$$

so

$$\mathbf{v}^{'} = \mathbf{T}_\theta \mathbf{v}_0. \tag{1.2}$$

On the other hand, consider the matrix that rotates the *coordinate system* through a counterclockwise angle θ. The coordinates of the fixed vector $\mathbf{v}$ in the rotated coordinate system are now given by a rotation matrix which is the *transpose* of the fixed-axis matrix, as can be seen on the
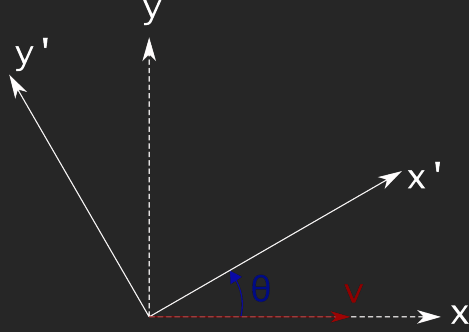
Figure 1.2: Rotating Axes

second figure, is equivalent to rotating the vector by a counterclockwise angle $-\theta$ relative to a fixed set of axes, giving:

$$\mathbf{T}'_{\theta} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}. \tag{1.3}$$

This is the convention commonly used in textbooks.

2. 3-D ($\mathbb{R}^3$):

In $\mathbb{R}^3$, coordinate system rotations of the x-, y- and z-axis in a counterclockwise direction when looking towards the origin give the matrices:

$$\mathbf{R}_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{bmatrix}. \tag{1.4}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}. \tag{1.5}$$

$$\mathbf{R}_z(\psi) = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.6}$$

Any **rotation** can be given as a composition of rotations about three axes (**Euler's rotation theorem**), and thus be represented by a 3×3 matrix operating on a vector.

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}. \tag{1.7}$$

3. **Euler's angles of XYZ rotation:**

   With the order of xyz rotation the rotation matrix is composed as:

$$\begin{aligned} \mathbf{T} &= \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_x(\varphi) \\ &= \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta\cos\psi & \cos\psi\sin\theta\sin\varphi - \cos\varphi\sin\psi & \cos\varphi\cos\psi\sin\theta + \sin\varphi\sin\psi \\ \cos\theta\sin\psi & \cos\varphi\cos\psi + \sin\theta\sin\varphi\sin\psi & \cos\varphi\sin\theta\sin\psi - \cos\psi\sin\varphi \\ -\sin\theta & \cos\theta\sin\varphi & \cos\theta\cos\varphi \end{bmatrix} \end{aligned} \tag{1.8}$$

   Then the three *Euler angles* for a xyz rotation can be obtained:

$$\varphi = \mathrm{atan2}(t_{32}, t_{33}), \tag{1.9}$$

$$\theta = \mathrm{atan2}\left(-t_{31}, \sqrt{t_{32}^2 + t_{33}^2}\right), \tag{1.10}$$

$$\psi = \mathrm{atan2}(t_{21}, t_{11}), \tag{1.11}$$

   where $\mathrm{atan2}(y, x)$ definition is:

$$\mathrm{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & if & (x > 0), \\ \arctan\left(\frac{y}{x}\right) + \pi & if & (x < 0) \wedge (y \geq 0), \\ \arctan\left(\frac{y}{x}\right) - \pi & if & (x < 0) \wedge (y < 0), \\ \frac{\pi}{2} & if & (x = 0) \wedge (y > 0), \\ -\frac{\pi}{2} & if & (x = 0) \wedge (y < 0), \\ undefined & if & (x = 0) \wedge (y = 0) \end{cases} . \tag{1.12}$$

Note the angle limitation for values returned while doing an Euler angle decomposition of a **rotation** matrix:

$$
\begin{aligned}
\varphi &\in (-\pi, \pi) \\
\theta &\in \left(-\tfrac{\pi}{2}, \tfrac{\pi}{2}\right) \\
\psi &\in (-\pi, \pi)
\end{aligned} \quad . \tag{1.13}
$$

4. Euler's angles of **ZYX** rotation:

   For a rotation defined in *zyx* order the rotation matrix is:

$$
\begin{aligned}
\mathbf{T} \quad &= \mathbf{R}_x(\varphi)\mathbf{R}_y(\theta)\mathbf{R}_z(\psi) \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi \\ 0 & -\sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta\cos\varphi & -\cos\theta\sin\varphi & \sin\varphi \\ \cos\psi\sin\varphi + \cos\varphi\sin\theta\sin\psi & \cos\varphi\cos\psi - \sin\theta\sin\varphi\sin\psi & -\cos\theta\sin\psi \\ \sin\varphi\sin\psi - \cos\varphi\cos\psi\sin\theta & \cos\psi\sin\theta\sin\varphi + \cos\varphi\sin\psi & \cos\theta\cos\psi \end{bmatrix}
\end{aligned} \quad . \tag{1.14}
$$

Note that the angles φ, θ and ψ defined here are different from the XYZ rotation defined above! The rotation is performed by first rotating around *z-axis* counterclockwise by an angle ψ, then around *y-axis* counterclockwise by an angle θ and finaly around *x-axis*, counterclockwise, by an angle φ.

Then the *Euler's angles* can be obtained as:

$$
\begin{cases}
\varphi = \arctan\left(\dfrac{-t_{12}}{t_{11}}\right) \\
\theta = \arcsin\left(t_{13}\right) \\
\psi = \arctan\left(\dfrac{-t_{23}}{t_{33}}\right)
\end{cases} \quad . \tag{1.15}
$$

5. Alternate way of constructing **Rotation matrix** without specifying angles:

   The rotation matrix to a new coordinate system can be constructed from its axes, where the rotation matrix can be defined as:

$$
\mathbf{T}_R = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix} , \tag{1.16}
$$

where $\mathbf{x} = [x_1, x_2, x_3]^T$, $\mathbf{y} = [y_1, y_2, y_3]^T$ and $\mathbf{z} = [z_1, z_2, z_3]^T$ are unit vectors in the direction of x-, y- and z-*axis* of the new coordinate system.

The **transformation matrix** can be defined by any two vectors $\mathbf{u}$ and $\mathbf{v}$, where $\mathbf{u}$ defines the direction of x-*axis* and $\mathbf{v}$ defines the *xy-plane*. To construct the **rotation matrix** use the following steps:

(a) create x-axis of unit lenght

$$\mathbf{x} = \frac{\mathbf{u}}{||\mathbf{u}||}. \tag{1.17}$$

(b) create z-axis of unit lenght

$$\mathbf{z} = \frac{\mathbf{u} \times \mathbf{v}}{||\mathbf{u} \times \mathbf{v}||}. \tag{1.18}$$

(c) create y-axis of unit lenght

$$\mathbf{y} = \frac{\mathbf{z} \times \mathbf{x}}{||\mathbf{z} \times \mathbf{x}||}. \tag{1.19}$$

(d) compose **rotation matrix**

$$\mathbf{T}_R = \begin{bmatrix} \mathbf{x}^T \\ \mathbf{y}^T \\ \mathbf{z}^T \end{bmatrix}. \tag{1.20}$$

The creation of the **rotation matrix** by any two other vectors is analogic, as **orthogonal** base must be constructed first, then the matrix is composed of its vectors.

To test, whether the transformation matrix is correct, one can use the orthogonality criterion:

$$\mathbf{T}^T = \mathbf{T}^{-1} \tag{1.21}$$

$$\mathbf{T}\mathbf{T}^T = \mathbf{T}^T\mathbf{T} = \mathbf{I} \tag{1.22}$$

or

$$\det(\mathbf{T}) = 1 \tag{1.23}$$

1.2 Coordinate Transformation

- **Rotation:**

Let $\mathbf{A}^T = [a_x, a_y, a_z]$ be the original coordinates, $\mathbf{B}^T = [b_x, b_y, b_z]$ be the transformed coordinates, and $\mathbf{T}$ a tranformation matrix in the form:

$$\mathbf{T} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix}, \tag{1.24}$$

where $\mathbf{x} = [x_1, x_2, x_3]$, $\mathbf{y} = [y_1, y_2, y_3]$ and $\mathbf{z} = [z_1, z_2, z_3]$ are the *x-axis*, *y-axis* and *z-axis* **unit vectors**, respectively, of the new coordinate system defined in the original one, while sharing their origin.

The following applies:

$$\mathbf{T} \times \mathbf{A} = \mathbf{B}, \tag{1.25}$$

and

$$\mathbf{T}^T \times \mathbf{B} = \mathbf{A}. \tag{1.26}$$

Note: Perpendicular vector is created using *cross-product*.

1.3   Tensor Transformation

From:

source 1, source 2, source 3

Let real symmetric matrix $\mathbf{A}$ be the original tensor, real symmetric matrix $\mathbf{B}$ the transformed tensor, and orthogonal matrix $\mathbf{T}$ a tranformation matrix in the form:

$$\mathbf{A} = \begin{bmatrix} \sigma_{11}^A & \sigma_{12}^A & \sigma_{13}^A \\ \sigma_{21}^A & \sigma_{22}^A & \sigma_{23}^A \\ \sigma_{31}^A & \sigma_{32}^A & \sigma_{33}^A \end{bmatrix}, \tag{1.27}$$

$$\mathbf{B} = \begin{bmatrix} \sigma_{11}^B & \sigma_{12}^B & \sigma_{13}^B \\ \sigma_{21}^B & \sigma_{22}^B & \sigma_{23}^B \\ \sigma_{31}^B & \sigma_{32}^B & \sigma_{33}^B \end{bmatrix}, \tag{1.28}$$

$$\mathbf{T} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix} = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix}, \tag{1.29}$$

where $\mathbf{x} = [x_1, x_2, x_3]$, $\mathbf{y} = [y_1, y_2, y_3]$ and $\mathbf{z} = [z_1, z_2, z_3]$ are the *x-axis*, *y-axis* and *z-axis* **unit vectors**, respectively, of the new coordinate system defined in the original one, while sharing their origin.

The transformation matrix $\mathbf{T}$ must satisfy $\mathbf{TT}^T = \mathbf{T}^T\mathbf{T} = \mathbf{I}$, where $\mathbf{I}$ is an identity matrix. This means that transformation matrix must be orthogonal, therefore satisfying $\mathbf{T}^T = \mathbf{T}^{-1}$.

Then **tensor transformation** is performed as such:

$$\mathbf{T} \times \mathbf{A} \times \mathbf{T}^T = \mathbf{B}. \tag{1.30}$$

$$\mathbf{T}^T \times \mathbf{B} \times \mathbf{T} = \mathbf{A}. \tag{1.31}$$

- **Principal values:**

  The principal values can be found as the **eigenvalues** of the tensor matrix.
  Characteristic equation of tensor $\mathbf{\Sigma}$:

  $$\mathbf{\Sigma V} = \mathbf{\Lambda V}, \tag{1.32}$$

  therefore:

  $$(\mathbf{\Sigma} - \mathbf{\Lambda I})\mathbf{V} = 0, \tag{1.33}$$

  and:

  $$det(\mathbf{\Sigma} - \mathbf{\Lambda I}) = 0, \tag{1.34}$$

  where $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues (principal values) and $\mathbf{V}$ is the eigenvector matrix.

  The characteristic cubic equation can be also written as:

  $$\lambda^3 - I_1\lambda^2 + I_2\lambda - I_3 = 0, \tag{1.35}$$

  giving three roots equal to tensor eigenvalues.

  First compute **Invariants $\mathbf{I}_n$**:

  $$\begin{aligned}
  I_1 &= \sigma_{11} + \sigma_{22} + \sigma_{33} \\
  I_2 &= \sigma_{11}\sigma_{22} + \sigma_{22}\sigma_{33} + \sigma_{33}\sigma_{11} - \sigma_{12}^2 - \sigma_{13}^2 - \sigma_{23}^2 \\
  I_3 &= \sigma_{11}\sigma_{22}\sigma_{33} - \sigma_{11}\sigma_{23}^2 - \sigma_{22}\sigma_{13}^2 - \sigma_{33}\sigma_{12}^2 + 2\sigma_{12}\sigma_{13}\sigma_{23}
  \end{aligned} \qquad . \tag{1.36}$$

  In matrix form:

  $$\begin{aligned}
  I_1 &= tr[\mathbf{\Sigma}] \\
  I_2 &= \begin{vmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{12} & \sigma_{22} \end{vmatrix} + \begin{vmatrix} \sigma_{11} & \sigma_{13} \\ \sigma_{13} & \sigma_{33} \end{vmatrix} + \begin{vmatrix} \sigma_{22} & \sigma_{23} \\ \sigma_{23} & \sigma_{33} \end{vmatrix} \\
  I_3 &= det(\mathbf{\Sigma})
  \end{aligned} \qquad . \tag{1.37}$$

  Then compute *help values*:

$$Q = \frac{3I_2 - I_1^2}{9}, \tag{1.38}$$

$$R = \frac{2I_1^3 - 9I_1 I_2 + 27I_3}{54}, \tag{1.39}$$

$$\theta = cos^{-1}\left(\frac{R}{\sqrt{-Q^3}}\right). \tag{1.40}$$

Lastly to get **principal values**:

$$\overline{\sigma_1} = 2\sqrt{-Q}\cos\left(\frac{\theta}{3}\right) + \frac{1}{3}I_1. \tag{1.41}$$

$$\overline{\sigma_2} = 2\sqrt{-Q}\cos\left(\frac{\theta + 2\pi}{3}\right) + \frac{1}{3}I_1. \tag{1.42}$$

$$\overline{\sigma_3} = 2\sqrt{-Q}\cos\left(\frac{\theta + 4\pi}{3}\right) + \frac{1}{3}I_1. \tag{1.43}$$

The **principal values are not sorted**. The principal tensor is then:

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix}, \tag{1.44}$$

where $\overline{\sigma}_1$, $\overline{\sigma}_2$ and $\overline{\sigma}_3 \implies \sigma_1 > \sigma_2 > \sigma_3$.

Afterwards the **principal axes** are obtained as:

$$\begin{aligned} \mathbf{\Gamma}_1 &= \mathbf{\Sigma} - \sigma_1 \mathbf{I} \\ \mathbf{\Gamma}_2 &= \mathbf{\Sigma} - \sigma_2 \mathbf{I} \\ \mathbf{\Gamma}_3 &= \mathbf{\Sigma} - \sigma_3 \mathbf{I} \end{aligned} \,, \tag{1.45}$$

where $\mathbf{\Gamma}$ is a coordinate system corresponding to the principal value i and $\mathbf{I}$ is a 3×3 identity matrix.

The vectors corresponding to each **principal value** are obtained:

$$\mathbf{\Gamma}_1 = \begin{bmatrix} \overline{x}_{11} & \overline{x}_{12} & \overline{x}_{13} \\ \overline{y}_{11} & \overline{y}_{12} & \overline{y}_{13} \\ \overline{z}_{11} & \overline{z}_{12} & \overline{z}_{13} \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{x}}_1 \\ \overline{\mathbf{y}}_1 \\ \overline{\mathbf{z}}_1 \end{bmatrix}, \tag{1.46}$$

$$\mathbf{\Gamma}_2 = \begin{bmatrix} \overline{x}_{21} & \overline{x}_{22} & \overline{x}_{23} \\ \overline{y}_{21} & \overline{y}_{22} & \overline{y}_{23} \\ \overline{z}_{21} & \overline{z}_{22} & \overline{z}_{23} \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{x}}_2 \\ \overline{\mathbf{y}}_2 \\ \overline{\mathbf{z}}_2 \end{bmatrix}, \tag{1.47}$$

$$\mathbf{\Gamma}_3 = \begin{bmatrix} \overline{x}_{31} & \overline{x}_{32} & \overline{x}_{33} \\ \overline{y}_{31} & \overline{y}_{32} & \overline{y}_{33} \\ \overline{z}_{31} & \overline{z}_{32} & \overline{z}_{33} \end{bmatrix} = \begin{bmatrix} \overline{\mathbf{x}}_3 \\ \overline{\mathbf{y}}_3 \\ \overline{\mathbf{z}}_3 \end{bmatrix}, \tag{1.48}$$

where $\overline{\mathbf{x}}$, $\overline{\mathbf{y}}$ and $\overline{\mathbf{z}}$ are vectors of size 3.

$$\mathbf{x} = \frac{\overline{\mathbf{y}}_1 \times \overline{\mathbf{z}}_1}{|\overline{\mathbf{y}}_1 \times \overline{\mathbf{z}}_1|}, \tag{1.49}$$

$$\mathbf{y} = \frac{\overline{\mathbf{z}}_2 \times \overline{\mathbf{x}}_2}{|\overline{\mathbf{z}}_2 \times \overline{\mathbf{x}}_2|}, \tag{1.50}$$

$$\mathbf{z} = \frac{\overline{\mathbf{x}}_3 \times \overline{\mathbf{y}}_3}{|\overline{\mathbf{x}}_3 \times \overline{\mathbf{y}}_3|}. \tag{1.51}$$

The principal axes are also the **eigenvectors** of tensor $\mathbf{\Sigma}$:

$$\mathbf{V} = \begin{bmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}. \tag{1.52}$$

- **Maximum Shear value:**

  Maximum **shear stress** occurs at an angle of 45 degrees to principal axes. If principal stresses are aligned such that $\sigma_1 > \sigma_2 > \sigma_3$ and the stress tensor is:

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix}, \tag{1.53}$$

  then maximum shear stress τ_{max} can be obtained as follows:

$$\tau_{max} = \frac{\sigma_1 - \sigma_3}{2} \tag{1.54}$$

and is at 45 degrees angle in the 1-3 plane of the principal axes to the principal coordinate system.

More generally (in 2D):

$$\tau_{max} = \sqrt{\left(\frac{\sigma_x - \sigma_y}{2}\right)^2 + \tau_{xy}^2}. \tag{1.55}$$

When the principal tensor is rotated by 45 degrees to obtain **maximum shear stress**, the axial stress components corresponding to this shear stress are equal.

The value of maximal shear stress is obtained:

$$\tau_{max} = \frac{\sigma_{max} - \sigma_{min}}{2}. \tag{1.56}$$

The value of axial stresses corresponding to maximal shear stress is the average of maximal and minimal axial stresses:

$$\sigma_{\tau_{max}} = \frac{\sigma_{max} + \sigma_{min}}{2}. \tag{1.57}$$

- **Example:**

  Let a stress tensor in principal coordinate system be:

$$\mathbf{\Sigma} = \begin{bmatrix} 433 & 0 & 0 \\ 0 & 125 & 0 \\ 0 & 0 & 24 \end{bmatrix}, \tag{1.58}$$

  then $\sigma_1 = \sigma_{max} = 433$ MPa and $\sigma_3 = \sigma_{min} = 24$ MPa.

  Transformation matrix for rotating by 45 degrees in 3-1 plane is:

$$\mathbf{T} = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix}. \tag{1.59}$$

Rotating tensor $\mathbf{\Sigma}$ by transformation matrix $\mathbf{T}$ we get:

$$\mathbf{\Sigma_\tau} = \mathbf{T\Sigma T}^T = \begin{bmatrix} 228.5 & 0 & 204.5 \\ 0 & 125 & 0 \\ 204.5 & 0 & 228.5 \end{bmatrix}, \quad (1.60)$$

where:

$$\begin{aligned} \tau_{max} &= \sigma_{13} = \sigma_{31} = 204.5 \\ \sigma_{11} &= \sigma_{33} = 228.5 \\ \sigma_{22} &= \sigma_2 \end{aligned} \quad . \quad (1.61)$$

Check for correct values:

$$\tau_{max} = \frac{\sigma_1 - \sigma_3}{2} = \frac{433 - 24}{2} = \frac{409}{2} = 204.5 \text{ MPa} \implies \text{OK!}, \quad (1.62)$$

$$\sigma_{11} = \sigma_{33} = \frac{\sigma_1 + \sigma_3}{2} = \frac{433 + 24}{2} = 228.5 \text{ MPa} \implies \text{OK!}. \quad (1.63)$$

$\boldsymbol{\sigma_2}$ **has not changed as it is colinear with the axis of rotation.**

The relationship between principal normal streses and maximum shear stresses can be better understood by examining a plot of the stresses as a function of the rotation angle.

Notice that there are multiple θ_p and $\theta_{\tau-max}$ angles because of the periodical nature of the equations. However, they will give the same absolute values.

At the principal stress angle, θ_p, the shear stress will always be zero, as shown on the diagram. And the maximum shear stress will occur when the two principal normal stresses, σ_1 and σ_2, are equal.

When σ_x or σ_y are either max or min, the shear stress τ_{xy} is equal to 0. When shear stress τ_{xy} is max or min, the stresses σ_x and σ_y are equal to their average.
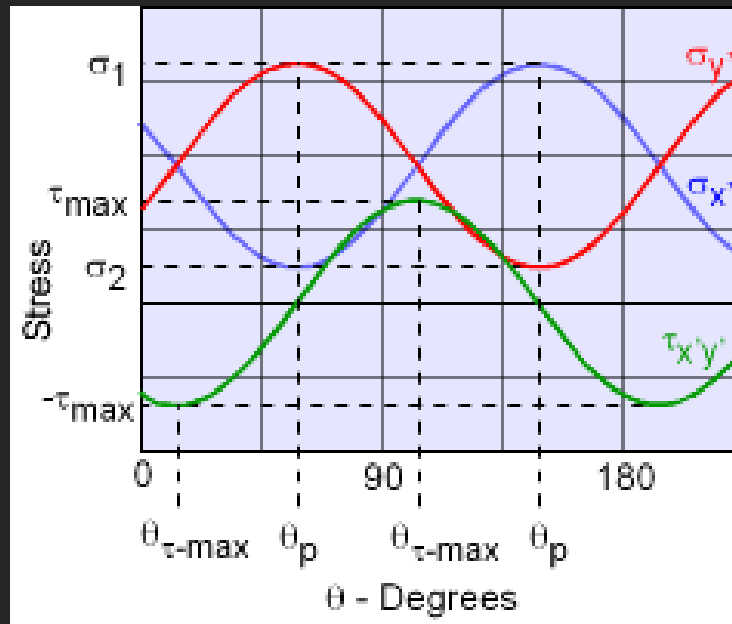
Figure 1.3: Stresses as a function of angle

1.4   Moment of Inertia

From: Parallel axis theorem

Parallel axis theorem, also known as *Huygens-Steiner's theorem* or just *Steiner's theorem* can be used to determine the **moment of inertia** or **second moment of area** of a rigid body about any axis, given the body's moment of inertia about a parallel axis through the objects center of gravity and the perpendicular distance between the axes.

Moment of Inertia = **tensor**.

- **Steiner's share:**

$$\mathbf{I}_{ss} = m \times \begin{bmatrix} y^2 + z^2 & -xy & -xz \\ -yz & x^2 + z^2 & -yz \\ -zx & -zy & x^2 + y^2 \end{bmatrix}. \tag{1.64}$$

Identities for a skew-symmetric matrix

In order to compare formulations of the parallel axis theorem using skew-symmetric matrices and the tensor formulation, the following identities are useful.

Let $[\mathbf{R}]$ be the skew-symmetric matrix associated with the position vector $\mathbf{R} = (x, y, z)$, then the product in the inertia matrix becomes:

$$-[\mathbf{R}][\mathbf{R}] = \begin{bmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{bmatrix}^2 = \begin{bmatrix} y^2 + z^2 & -xy & -xz \\ -yz & x^2 + z^2 & -yz \\ -zx & -zy & x^2 + y^2 \end{bmatrix}. \tag{1.65}$$

This product can be computed using the matrix formed by the outer product $[\mathbf{R}\mathbf{R}^T]$ using the identity

$$\begin{aligned} -[\mathbf{R}]^2 &= |\mathbf{R}|^2 |\mathbf{E}_3| - [\mathbf{R}\mathbf{R}^T] \\ &= \begin{bmatrix} x^2 + y^2 + z^2 & 0 & 0 \\ 0 & x^2 + y^2 + z^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} x^2 & xy & xz \\ yx & y^2 & yz \\ zx & zy & z^2 \end{bmatrix} \end{aligned}, \tag{1.66}$$

where $[\mathbf{E}_3$ is the $n \times n$ identity matrix.

Also notice, that

$$|\mathbf{R}|^2 = \mathbf{R} \cdot \mathbf{R} = tr[\mathbf{R}\mathbf{R}^T]. \tag{1.67}$$

where tr denotes the sum of the diagonal elements of the outer product matrix, known as its *trace*.

To obtain matrix of moments of inertia by rotating about any random point A while having the moment of inertia in the center of gravity of the object $\mathbf{I}_{COG}$ and the coordinate distance $d = [x, y, z]$ between the point A and **COG** we:

1. Compute the moment of inertia relative to point A

$$\mathbf{I}_A = \mathbf{I}_{COG} + m \times \mathbf{I}_{SS}^{A \to COG}. \tag{1.68}$$

2. Transform the tensor $\mathbf{I}_A$ to new rotated coordinate system using transformation matrix $\mathbf{T}$:

$$\mathbf{I}_A^{'} = \mathbf{T} \times \mathbf{I}_A \times \mathbf{T}^T. \tag{1.69}$$

3. Subtract the **Steiner's share** relative to new **COG**

$$\mathbf{I}_{COG}^{'} = \mathbf{I}_A^{'} - \mathbf{I}_{SS}^{A' \to COG'}. \tag{1.70}$$

1.5   Quaternions

In mathematics, the quaternion number system extends the complex numbers. Quaternions were first described by the Irish mathematician William Rowan Hamilton in 1843 and applied to mechanics in three-dimensional space. The algebra of quaternions is often denoted by $\mathbf{H}$ or $\mathbb{H}$ (for Hamilton).

Although multiplication of quaternions is noncommutative, it gives a definition of the quotient of two vectors in a three-dimensional space. Quaternions are generally represented in the form:

$$a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k} \tag{1.71}$$

where the coefficients a, b, c, d are *real numbers* and $\mathbf{1}, \mathbf{i}, \mathbf{j}, \mathbf{k}$ are *basis vectors* or *basis elements*.

The Quaterninon multiplication table (left column shows pre-multiplier, top row shows post-multiplier):

r × c	1	i	j	k
1	1	i	j	k
i	i	−1	k	−j
j	j	−k	−1	i
k	k	j	−i	−1

Table 1.1: Quaternion multiplication table

also, $a\mathbf{b} = \mathbf{b}a$ and $-\mathbf{b} = (-1)\mathbf{b}$ for $a \in \mathbb{R}, \quad b = \mathbf{i}, \mathbf{j}, \mathbf{k}$.

1.5.1   Quaternion properties

The following properties apply:

1. Addition:

$$(a_1 + b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k}) + (a_2 + b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k}) =$$
$$= (a_1 + a_2) + (b_1 + b_2)\mathbf{i} + (c_1 + c_2)\mathbf{j} + (d_1 + d_2)\mathbf{k} \qquad (1.72)$$

2. Component-wise scalar multiplication:

$$\lambda(a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}) = \lambda a + (\lambda b)\mathbf{i} + (\lambda c)\mathbf{j} + (\lambda d)\mathbf{k} \qquad (1.73)$$

3. Multiplication of basis elements (the revelation of Hamilton in 1943):

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = -1$$
$$\mathbf{ijk} = -1 \qquad (1.74)$$

4. Multiplicative identity:

$$
\begin{aligned}
\mathbf{i}1 &= \quad 1\mathbf{i} = \quad \mathbf{i} \\
\mathbf{j}1 &= \quad 1\mathbf{j} = \quad \mathbf{j} \\
\mathbf{k}1 &= \quad 1\mathbf{k} = \quad \mathbf{k}
\end{aligned}
\qquad (1.75)
$$

5. Products of basis elements:

$$
\begin{aligned}
\mathbf{ij} &= -\mathbf{ji} = \quad \mathbf{k} \\
\mathbf{jk} &= -\mathbf{kj} = \quad \mathbf{i} \\
\mathbf{ki} &= -\mathbf{ik} = \quad \mathbf{j}
\end{aligned}
\qquad (1.76)
$$

6. Inverse:

$$(a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k})^{-1} = \frac{1}{a^2 + b^2 + c^3 + d^2}(a - b\mathbf{i} - c\mathbf{j} - d\mathbf{k}) \qquad (1.77)$$

7. Product of two **quaternions** (also called the **Hamilton product**) is determined by the products of the basis elements and the **distributive law**:

$$(a_1 + b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k}) \times (a_2 + b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k}) = \qquad (1.78)$$

$$
\begin{aligned}
= & a_1 a_2 & + a_1 b_2 \mathbf{i} & \quad + a_1 c_2 \mathbf{j} & \quad + a_1 d_2 \mathbf{k} \\
+ & b_1 a_2 \mathbf{i} & + b_1 b_2 \mathbf{i}^2 & \quad + b_1 c_2 \mathbf{ij} & \quad + b_1 d_2 \mathbf{ik} \\
+ & c_1 a_2 \mathbf{j} & + c_1 b_2 \mathbf{ij} & \quad + c_1 c_2 \mathbf{j}^2 & \quad + c_1 d_2 \mathbf{jk} \\
+ & d_1 a_2 \mathbf{k} & + d_1 b_2 \mathbf{ik} & \quad + d_1 c_2 \mathbf{jk} & \quad + d_1 d_2 \mathbf{k}^2
\end{aligned}
\qquad (1.79)
$$

$$
\begin{aligned}
= & a_1 a_2 & - b_1 b_2 & \quad - c_1 c_2 & \quad - d_1 d_2 \\
+ & (a_1 b_2 & + b_1 a_2 & \quad + c_1 d_2 & \quad - d_1 c_2) \times \mathbf{i} \\
+ & (a_1 c_2 & - b_1 d_2 & \quad + c_1 a_2 & \quad + d_1 b_2) \times \mathbf{j} \\
+ & (a_1 d_2 & + b_1 c_2 & \quad - c_1 b_2 & \quad + d_1 a_2) \times \mathbf{k}
\end{aligned}
\qquad (1.80)
$$

1.5.2   Scalar and vector parts

A quaternion of the form $a + 0\mathbf{i} + 0\mathbf{j} + 0\mathbf{k}$, where a is a real number, is called **scalar**, and a quaternion of the form $0 + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ where b, c and d are real numbers, and at least on of b, c or d is nonzero, is called a **vector quaternion**. If $a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ is any quaternion, then a is called its **scalar part** and $b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ is called its **vector part**. Although every quaternion can be viewed as a vector in a four-dimensional vector space, it is common to refer to the **vector** part as vectors in three-dimensional space.

Hamilton also called vector quaternions **right quaternions** and real numbers (considered as quaternions with zero vector part) **scalar quaternions**.

If a quaternion is divided up into a scalar and a vector part, that is,

$$\mathbf{q} = (r, \vec{v}), \mathbf{q} \in \mathbb{H}, r \in \mathbb{R}, \vec{v} \in \mathbb{R}^3 \qquad (1.81)$$

then the formulas for **addition**, **multiplication** and **multiplicative inverse** are:

$$(r_1, \vec{v}_1) + (r_2, \vec{v}_2) = (r_1 + r_2, \vec{v}_1 + \vec{v}_2) \qquad (1.82)$$

$$(r_1, \vec{v}_1) \times (r_2, \vec{v}_2) = (r_1 r_2 - \vec{v}_1 \cdot \vec{v}_2, r_1 \vec{v}_2 + r_2 \vec{v}_1 + \vec{v}_1 \times \vec{v}_2) \qquad (1.83)$$

$$(r, \vec{v})^{-1} = \left(\frac{r}{r^2 + \vec{v} \cdot \vec{v}}, \frac{-\vec{v}}{r^2 + \vec{v} \cdot \vec{v}} \right) \qquad (1.84)$$

1.5.3   Conjugation, the norm and reciprocal

Conjugation of quaternions is analogous to conjugation of complex numbers. To define it, let $\mathbf{q} = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ be a quaternion. The **conjugate** of $\mathbf{q}$ is the quaternion $\mathbf{q}^* = a - b\mathbf{i} - c\mathbf{j} - d\mathbf{k}$. It is denoted by $\mathbf{q}^*$, $\mathbf{q}^t$ or $\bar{\mathbf{q}}$.

Conjugation is **involution**, meaning that it is its own inverse, so **conjugating** an element **twice** returns the original element. The **conjugate** of a **product** of two quaternions is the product of the conjugates in the **reverse order**. That is, if $\mathbf{p}$ and $\mathbf{q}$ are quaterninos, then:

$$(\mathbf{pq})^* = \mathbf{q}^* \mathbf{p}^* \qquad (1.85)$$

The conjugation of a quaternion, in stark contrast to the complex setting, can be expressed with multiplication and addition of quaternions:

$$\mathbf{q}^* = -\frac{1}{2}(\mathbf{q} + \mathbf{iqi} + \mathbf{jqj} + \mathbf{kqi}) \qquad (1.86)$$

Conjugation can be used to extract the **scalar** and **vector** part of a quaternion. The **scalar** part of $\mathbf{q}$ is:

$$r_q = \frac{1}{2}(\mathbf{q} + \mathbf{q}^*) \qquad (1.87)$$

The **vector** part of $\mathbf{q}$ is:

$$\vec{v}_q = \frac{1}{2}(\mathbf{q} - \mathbf{q}^*) \qquad (1.88)$$

The **square root** of the product of a quaternion with its conjugate is called its **norm** and is denoted $\|q\|$ (Hamilton called this quantity the **tensor** of $\mathbf{q}$).

$$\|\mathbf{q}\| = \sqrt{\mathbf{q}^*\mathbf{q}} = \sqrt{\mathbf{q}\mathbf{q}^*} = \sqrt{a^2 + b^2 + c^2 + d^2} \in \mathbb{R} \geq 0 \qquad (1.89)$$

This is always a non-negative, real number and is the same as the Euclidean norm on $\mathbb{H}$ considered as the vector space $\mathbb{R}^4$. Multiplying a quaternion by a real number scales its norm by the absolute value of the number. That is, if $\alpha \in \mathbb{R}$ then:

$$\|\alpha\mathbf{q}\| = |\alpha|\|\mathbf{q}\| \qquad (1.90)$$

The norm is **multiplicative**, meaning that:

$$\|\mathbf{p}\mathbf{q}\| = \|\mathbf{p}\|\|\mathbf{q}\| \qquad (1.91)$$

for any two quaternion $\mathbf{p}$ and $\mathbf{q}$. Multiplicativity is a consequence of the formula for the conjugate of a product. Alternatively it follows fom the identity

$$det \begin{pmatrix} a + ib & id + c \\ id - c & a - ib \end{pmatrix} = a^2 + b^2 + c^2 + d^2 \qquad (1.92)$$

(where i denotes the usual imaginary unit) and hence from the multiplicative property of determinants of square matrices.

This norm makes it possible to define the **distance** $d(\mathbf{p}, \mathbf{q})$ between $\mathbf{p}$ and $\mathbf{q}$ as the norm of their difference:

$$d(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\| \qquad (1.93)$$

This makes $\mathbb{H}$ a **metric space**. Addition and multiplication are **continuous** in regard to the associated metric topology. This follows exactly the same proof as for the real numbers $\mathbb{R}$ from the fact that $\mathbb{H}$ is a **normed algebra**.

1.5.4   Unit quaternion (or versor)

A **unit quaternion** is a quaternion of norm one. Dividing a nonzero quaternion **q** by its norm produces a unit quaternion **Uq** called the **versor** of **q**:

$$\mathbf{Uq} = \frac{\mathbf{q}}{\|\mathbf{q}\|} \tag{1.94}$$

Every nonzero quaternion has a unique **polar decomposition** $\mathbf{q} = \|\mathbf{q}\| \cdot \mathbf{Uq}$, while the zero quaternion can be formed from any unit quaternion.

Using conjugation and the norm makes it possible to define the **reciprocal** of a nonzero quaternion. The product of a quaternion with its reciprocal should be equal 1, and the considerations above imply that the product of **q** and $\mathbf{q}^*/\|\mathbf{q}\|^2$ is 1 (for any order fo multiplication). So the **reciprocal** of **q** is defined to be:

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{\|\mathbf{q}\|^2} \tag{1.95}$$

Since the multiplication is non-commutative, the quotient quantities $\mathbf{pq}^{-1}$ or $\mathbf{q}^{-1}\mathbf{p}$ are different (except if **p** and **q** are scalar multiples of each other or if one is a scalar). The notation $\frac{p}{q}$ is ambiguous and **should not be used**.

1.5.5   Quaternions and three-dimensional geometry

The vector part of a quaternion can be interpreted as a coordinate vector in $\mathbb{R}^3$, therefore the algebraic operations fo the quaternions reflect the geometry of $\mathbb{R}^3$. Operations such as the vector dot and cross products can be defined in terms of quaternions and thei makes it possible to apply quaternion techniques wherever spatial vectors arise. A useful application of quaternions has been to interpolate the orientations of key-frames in computer graphics.

For the remainder of this section, **i**, **j** and **k** wil denote both the three imaginary basis vectors of $\mathbb{H}$ and a basis for $\mathbb{R}^3$. Replacing **i** by $-\mathbf{i}$, **j** by $-\mathbf{j}$ and **k** by $-\mathbf{k}$ sends a vector to its **additive inverse**, so the additive inverse of a vector is the

same as its conjugate as a quaternion. For this reason, conjugation is sometimes called the **spatial inverse**.

For two vector quaternions $\mathbf{p} = b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k}$ and $\mathbf{q} = b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k}$ their **dot product**, by analogy to vectors in $\mathbb{R}^3$, is:

$$\mathbf{p} \cdot \mathbf{q} = b_1 b_2 + c_1 c_2 + d_1 d_2 \tag{1.96}$$

It can also be expressed ina a component-free manner as:

$$\mathbf{p} \cdot \mathbf{q} = \frac{1}{2}(\mathbf{p}^*\mathbf{q} + \mathbf{q}^*\mathbf{p}) = \frac{1}{2}(\mathbf{p}\mathbf{q}^* + \mathbf{q}\mathbf{p}^*) \tag{1.97}$$

This is equal to the scalar parts of the products $\mathbf{p}\mathbf{q}^*$, $\mathbf{q}\mathbf{p}^*$, $\mathbf{p}^*\mathbf{q}$ and $\mathbf{q}^*\mathbf{p}$. Note that their vector parts are different.

The **cross product** of $\mathbf{p}$ and $\mathbf{q}$ relative to the orientation determined by the ordered basis $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ is:

$$\mathbf{p} \times \mathbf{q} = (c_1 d_2 - d_1 c_2)\mathbf{i} + (d_1 b_2 - b_1 d_2)\mathbf{j} + (b_1 c_2 - c_1 b_2)\mathbf{k} \tag{1.98}$$

(The orientation is necessary to determine the sign.) This is equal to the vector part of the product $\mathbf{p}\mathbf{q}$ (as quaternions), as well as the vector part of $-\mathbf{q}^*\mathbf{p}^*$. It also has the formula:

$$\mathbf{p} \times \mathbf{q} = \frac{1}{2}(\mathbf{p}\mathbf{q} - \mathbf{q}\mathbf{p}) \tag{1.99}$$

For the **commutator**, $[\mathbf{p}, \mathbf{q}] = \mathbf{p}\mathbf{q} - \mathbf{q}\mathbf{p}$ of two vector quaternions one obtains:

$$[\mathbf{p}, \mathbf{q}] = 2\mathbf{p} \times \mathbf{q} \tag{1.100}$$

In general, let $\mathbf{p}$ and $\mathbf{q}$ be quaternions and write:

$$\mathbf{p} = \mathbf{p}_s + \mathbf{p}_v$$
$$\mathbf{q} = \mathbf{q}_s + \mathbf{q}_v \tag{1.101}$$

where $\mathbf{p}_s$ and $\mathbf{q}_s$ are the scalar parts, and $\mathbf{p}_v$ and $\mathbf{q}_v$. Then we have the formula:

$$\mathbf{pq} = (\mathbf{pq})_s + (\mathbf{pq})_v = (\mathbf{p}_s\mathbf{q}_s - \mathbf{p}_v \cdot \mathbf{q}_v) + (\mathbf{p}_s\mathbf{q}_v + \mathbf{q}_s\mathbf{p}_v + \mathbf{p}_v \times \mathbf{q}_v) \tag{1.102}$$

This shows that the noncommutativity of quaternion multiplication comes from the multiplication of vector quaternions. It also shows that two quaternions commute if and only if their vector parts are collinear. **Hamilton** showed that this product computes the third vertex of spherical triangle from two given vertices and their associated arc-lengths, which is also an algebra of points in **Elliptic geometry**.

Unit quaternions can be identified with rotations in $\mathbb{R}^3$ and were called **versors** by **Hamilton**.

1.5.6 Matrix representations

Just as complex numbers can be **represented as matrices**, co can be quaternions. There are *at least* two ways to represent quaternions as matrices in such a way that quaternion **addition** and **multiplication** correspond to the matrix addition and multiplication.

The quaternion $a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ can be represented:

1. as a 2×2 complex matrix:

$$\begin{bmatrix} a + bi & c + di \\ -c + di & a - bi \end{bmatrix} \tag{1.103}$$

   This representation has the following properties:

- Constraining any two of b, c and d to zero produces a representation fo complex numbers. For example, setting $c = d = 0$ produces a diagonal complex matrix representation of complex numbers, and setting $b = d = 0$ produces a real matrix representation.
- The **norm** of a quaternion (the square root of the product with its conjugate, as with complex numbers) is the square root of the **determinant** of the corresponding matrix.
- The **conjugate** of a quaternion corresponds to the **conjugate transpose** of the matrix.
- There is a strong relation between quaternion units and **Pauli matrices** (these are needed to describe the *spin in quantum mechanics*). Obtain the eight quaternion unit matrices by taking a, b, c and d, set three of them to zero and the fourth to 1 or -1. Multiplying any two Pauli matrices always yields a quaternion unit matrix, all of them except for -1. One obtains -1 via the last equality:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$
$$ijk = \sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_3 = -1 \tag{1.104}$$

2. Using 4×4 real matrix the same quaternion can be written as:

$$
\begin{bmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{bmatrix} = a \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$
$$
+ b \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix}
$$
$$
+ c \begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \tag{1.105}
$$
$$
+ d \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}
$$

However the representation above is not unique. For example, the same quaternion can be also represented as:

$$
\begin{bmatrix}
a & d & -b & -c \\
-d & a & c & -b \\
b & -c & a & -d \\
c & b & d & a
\end{bmatrix}
= a
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$
$$
+ b
\begin{bmatrix}
0 & 0 & -1 & 0 \\
0 & 0 & 0 & -1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
$$
$$
+ c
\begin{bmatrix}
0 & 0 & 0 & -1 \\
0 & 0 & 1 & 0 \\
0 & -1 & 0 & 0 \\
1 & 0 & 0 & 0
\end{bmatrix}
$$
$$
+ d
\begin{bmatrix}
0 & 1 & 0 & 0 \\
-1 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 \\
0 & 0 & 1 & 0
\end{bmatrix}
\tag{1.106}
$$

There exist 48 distinct matrix representations of thei form in which one of the matrices represents the scalar part and the other three are all skew-symmetric.

1.5.7   Exponential, logarithm and power functions

Givena quaternion:

$$
\mathbf{q} = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k} = r + \vec{\mathbf{v}}
\tag{1.107}
$$

1. the exponential is computed as

$$
\exp(\mathbf{q}) = \sum_{n=0}^{\infty} \frac{\mathbf{q}^n}{n!} = e^a \left(\cos \|\vec{\mathbf{v}}\| + \frac{\vec{\mathbf{v}}}{\|\vec{\mathbf{v}}\|} \sin \|\vec{\mathbf{v}}\| \right)
\tag{1.108}
$$

2. the logarithm is:

$$\ln(\mathbf{q}) = \ln \|\mathbf{q}\| + \frac{\vec{\mathbf{v}}}{\|\vec{\mathbf{v}}\|} \arccos \frac{r}{\|\mathbf{q}\|} \tag{1.109}$$

3. it follows that the polar decomposition of a quaternion may be written

$$\mathbf{q} = \|\mathbf{q}\| e^{\hat{\mathbf{n}}\varphi} = \|\mathbf{q}\| \left(\cos(\varphi) + \hat{\mathbf{n}} \sin(\varphi) \right) \tag{1.110}$$

where the angle φ is defined as

$$r = \|\mathbf{q}\| \cos(\varphi) \tag{1.111}$$

and the unit vecotr $\hat{\mathbf{n}}$ is defined by

$$\vec{\mathbf{v}} = \hat{\mathbf{n}} \|\vec{\mathbf{v}}\| = \hat{\mathbf{n}} \|\mathbf{q}\| \sin(\varphi) \tag{1.112}$$

Any unit quaternion may be expressed in polar form as:

$$\mathbf{q} = \exp(\hat{\mathbf{n}}\varphi) \tag{1.113}$$

4. The power of a quaternion raised to an arbitrary (real) exponend x is given by:

$$\mathbf{q}^x = \|\mathbf{q}\|^x e^{\hat{\mathbf{n}}x\varphi} = \|\mathbf{q}\|^x \left(\cos(x\varphi) + \hat{\mathbf{n}} \sin(x\varphi) \right) \tag{1.114}$$

1.5.8   Quaternions used for coordinate transformation

One can think of **quaternions** as a stored information about coordinate mapping from one space to another. We can split arbitrary **quaternion** $\mathbf{q} = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ into **two** parts:

(a) a **real** part $r = a$ and
(b) a **vector** part $\mathbf{v} = [b, c, d]$.

Next, if we **impose** restrictions on such **quaternions** that:

(a) the **real** part must satisfy: $r \in \langle -\pi; \pi \rangle$

(b) the **vector** part must have **unit** length: $|\mathbf{v}| = \sqrt{b^2 + c^2 + d^2} = 0$

Then, if one thinks of the complex numbers $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ as of basis vectors of cartesian system $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$, where

$$
\begin{aligned}
\mathbf{x} &= [1.0, 0.0, 0.0] \\
\mathbf{y} &= [0.0, 1.0, 0.0] \\
\mathbf{z} &= [0.0, 0.0, 1.0]
\end{aligned}
\tag{1.115}
$$

the quaternion multiplication table 1.1 makes sense in relation to **vector cross product**:

$r \times c$	1	x	y	z
1	1	x	y	z
x	x	−1	z	−y
y	y	−z	−1	x
z	z	y	−x	−1

Table 1.2: Quaternion cartesian base vectors multiplication table

Chapter 2

Boundary conditions not in GCS

2.1   Skew boundary conditions

Skew boundary conditions, also known as **Rotated BCs** are implemented after assembling the global stiffness matrix $\mathbf{K}$.

Basic equation:

$$\mathbf{M\ddot{u}} + \mathbf{C\dot{u}} + \mathbf{Ku} = \mathbf{f} \tag{2.1}$$

First we need to identify all prescribed BCs that do not correspond to the defined assemblage DOFs and transform the finite equilibrium equations to correspond to the prescribed BCs. Thus we write:

$$\mathbf{u} = \mathbf{T\bar{u}} \tag{2.2}$$

where $\mathbf{\bar{u}}$ is the vector of nodal point DOFs in the required directions. The transformation matrix $\mathbf{T}$ is an identity matrix that has been altered by the directional sines and cosines of the components $\mathbf{\bar{u}}$ measured in the original displacement directions. Using (2.2), we obtain:

$$\overline{\mathbf{M}}\ddot{\overline{\mathbf{u}}} + \overline{\mathbf{C}}\dot{\overline{\mathbf{u}}} + \overline{\mathbf{K}}\overline{\mathbf{u}} = \overline{\mathbf{f}} \tag{2.3}$$

where:

$$
\begin{aligned}
\overline{\mathbf{M}} &= \mathbf{T}^T \mathbf{M} \mathbf{T} \\
\overline{\mathbf{C}} &= \mathbf{T}^T \mathbf{C} \mathbf{T} \\
\overline{\mathbf{K}} &= \mathbf{T}^T \mathbf{K} \mathbf{T} \\
\overline{\mathbf{f}} &= \mathbf{T}^T \mathbf{f}
\end{aligned}
\tag{2.4}
$$

We should note that the matrix multiplications in (2.4) involve changes only in those columns and rows of $\mathbf{M}$, $\mathbf{C}$, $\mathbf{K}$ and $\mathbf{f}$ that are actually affected. In practice, the transformation is carried out effectively on the element level just prior to adding the element matrices to the global matrices.

2.2   Cartesian CSYS

When a node or an SPC is defined as a **rotated**/**skewed** or in a different type of Coordinate system (e.g. cylindrical) a transformation matrix needs to be applied to such nodes.

Cartesian Coordinate system definition by angle (2D):

$$\mathbf{u}_l = \mathbf{T}\mathbf{u}_g \tag{2.5}$$

Unpacked:

$$\begin{bmatrix} u_l \\ w_l \\ \psi_l \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_g \\ w_g \\ \psi_g \end{bmatrix} \tag{2.6}$$

Cartesian Coordinate system definition by vectors (3D):

when a **CSYS** is defined by an **origin** and **two vectors** $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}$ and $\mathbf{y} = \begin{pmatrix} y_1 & y_2 & y_3 \end{pmatrix}$, then for a Cartesian CSYS it suffices to:

$$\begin{aligned} \bar{\mathbf{x}} &= \frac{\mathbf{x}}{||\mathbf{x}||} \\ \bar{\mathbf{y}} &= \frac{\mathbf{y}}{||\mathbf{y}||} \\ \bar{\mathbf{z}} &= \bar{\mathbf{x}} \times \bar{\mathbf{y}} \end{aligned} \tag{2.7}$$

The transformation relation:

$$\mathbf{u}_l = \mathbf{T}\mathbf{u}_g \tag{2.8}$$

when unpacked:

$$\begin{bmatrix} u_l \\ v_l \\ w_l \\ \phi_l \\ \psi_l \\ \rho_l \end{bmatrix} = \begin{bmatrix} \mathbf{x} & \mathbf{0} \\ \mathbf{y} & \mathbf{0} \\ \mathbf{z} & \mathbf{0} \\ \mathbf{0} & \mathbf{x} \\ \mathbf{0} & \mathbf{y} \\ \mathbf{0} & \mathbf{z} \end{bmatrix} \begin{bmatrix} u_g \\ v_g \\ w_g \\ \phi_g \\ \psi_g \\ \rho_g \end{bmatrix} \tag{2.9}$$

Where $\mathbf{0}$ is a 3×1 zero vector.

For a **2D case** just **one vector** is enough, the $\bar{\mathbf{z}} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$ and following relations apply:

$$\begin{aligned} \bar{\mathbf{x}} &= \bar{\mathbf{y}} \times \bar{\mathbf{z}} \\ \bar{\mathbf{y}} &= \bar{\mathbf{z}} \times \bar{\mathbf{x}} \\ \bar{\mathbf{z}} &= \bar{\mathbf{x}} \times \bar{\mathbf{y}} \end{aligned} \qquad (2.10)$$

This means that a **CSYS** defined by an **origin** point $\mathbf{O} = \begin{pmatrix} x_O & y_O & z_O \end{pmatrix}$, point laying on **x axis** $\mathbf{P}_x = \begin{pmatrix} x_{P_x} & y_{P_x} & z_{P_x} \end{pmatrix}$ and a point laying in **xy plane** $\mathbf{P}_{xy} = \begin{pmatrix} x_{P_{xy}} & y_{P_{xy}} & z_{P_{xy}} \end{pmatrix}$ define a CSYS followingly:

$$\begin{aligned} \mathbf{x} &= \begin{pmatrix} x_{P_x} - x_O & y_{P_x} - y_O & z_{P_x} - z_O \end{pmatrix} \\ \bar{\mathbf{x}} &= \frac{\mathbf{x}}{||\mathbf{x}||} \\ \hat{\mathbf{y}} &= \begin{pmatrix} x_{P_{xy}} - x_O & y_{P_{xy}} - y_O & z_{P_{xy}} - z_O \end{pmatrix} \\ \bar{\hat{\mathbf{y}}} &= \frac{\hat{\mathbf{y}}}{||\hat{\mathbf{y}}||} \\ \bar{\mathbf{z}} &= \bar{\mathbf{x}} \times \bar{\hat{\mathbf{y}}} \\ \bar{\mathbf{y}} &= \bar{\mathbf{z}} \times \bar{\mathbf{x}} \end{aligned} \qquad (2.11)$$

2.3   Cylindrical CSYS

The basic relation between a point in **cartesian** and **cylindrical** CSYS
is (when the cartesian CSYS has the same origin as the cylindrical one
and both CSYS z axes overlay):

$$x = r \cos \varphi$$
$$y = r \sin \varphi \qquad\qquad (2.12)$$
$$z = z$$

in one direction and:

$$r = \sqrt{x^2 + y^2}$$

$$\varphi = \begin{cases} indeterminate & if\ x = 0\ and\ y = 0 \\ \arcsin \frac{y}{r} & if\ x \geq 0 \\ -\arcsin \frac{y}{r} + \pi & if\ x < 0\ and\ y \geq 0 \\ -\arcsin \frac{y}{r} + \pi & if\ x < 0\ and\ y < 0 \end{cases} \qquad (2.13)$$

in the other. One can also use the arctan function to compute φ:

$$\varphi = \begin{cases} indeterminate & if\ x = 0\ and\ y = 0 \\ \frac{\pi}{2} \frac{y}{|y|} & if\ x = 0\ and\ y \neq 0 \\ \arctan \frac{y}{x} & if\ x > 0 \\ \arctan \frac{y}{x} + \pi & if\ x < 0\ and\ y \geq 0 \\ \arctan \frac{y}{x} - \pi & if\ x < 0\ and\ y < 0 \end{cases} \qquad (2.14)$$

This is also called the $\text{atan2}(y, x)$ function!

To transform node coordinates from **cartesian** to **cylindrical**, three trans-
forms are in order:

(a) translate the coordinates so that **origin** of the new CSYS conforms
    to the **origin** of the **cylidrical** CSYS.

$$\mathbf{x}_1 = \mathbf{T}_T \mathbf{x}_0 \qquad\qquad (2.15)$$

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -x_O^c \\ 0 & 1 & 0 & -y_O^c \\ 0 & 0 & 1 & -z_O^c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \qquad (2.16)$$

where $\mathbf{x}_O^c$ denotes the **cylindrical** CSYS origin and $\mathbf{T}_T$ denotes the translation matrix.

(b) **rotate** the node to a new **cartesian** CSYS so that the new CSYS **x-axis** conforms to the cylindrical CSYS **r-axis** and the new CSYS **z-axis** conforms to the cylindrical **z-axis**.

$$\mathbf{x}_2 = \mathbf{T}_R \mathbf{x}_1 \tag{2.17}$$

When the cylindrical CSYS is defined by three points - **origin** $\mathbf{x}_O^c = \begin{bmatrix} x_O^C & y_O^c & z_O^c \end{bmatrix}^T$, point on **r-axis** $\mathbf{x}_R^c$ and a point in **rz-plane** $\mathbf{x}_{RZ}^c$, then:

$$\vec{\mathbf{r}} = \frac{\mathbf{x}_R^c - \mathbf{x}_O^c}{||\mathbf{x}_R^c - \mathbf{x}_O^c||} \tag{2.18}$$

$$\hat{\mathbf{z}} = \frac{\mathbf{x}_{RZ}^c - \mathbf{x}_O^c}{||\mathbf{x}_{RZ}^c - \mathbf{x}_O^c||} \tag{2.19}$$

$$\vec{\mathbf{y}} = \hat{\mathbf{z}} \times \vec{\mathbf{r}} \tag{2.20}$$

$$\vec{\mathbf{z}} = \vec{\mathbf{r}} \times \vec{\mathbf{y}} \tag{2.21}$$

then the transformation can be written:

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{r}}^T & 0 \\ \vec{\mathbf{y}}^T & 0 \\ \vec{\mathbf{z}}^T & 0 \\ \vec{\mathbf{0}}^T & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \tag{2.22}$$

where $\vec{\mathbf{0}} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$

Combining $\mathbf{T}_R$ and $\mathbf{T}_T$ together:

$$\mathbf{T} = \mathbf{T}_R \mathbf{T}_T \tag{2.23}$$

$$\mathbf{T} = \begin{bmatrix} r_1 & r_2 & r_3 & 0 \\ y_1 & y_2 & y_3 & 0 \\ z_1 & z_2 & z_3 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_O^c \\ 0 & 1 & 0 & -y_O^c \\ 0 & 0 & 1 & -z_O^c \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.24}$$

$$\mathbf{T} = \begin{bmatrix} r_1 & r_2 & r_3 & -r_1 x_O^c - r_2 y_O^c - r_3 z_O^c \\ y_1 & y_2 & y_3 & -y_1 x_O^c - y_2 y_O^c - y_3 z_O^c \\ z_1 & z_2 & z_3 & -z_1 x_O^c - z_2 y_O^c - z_3 z_O^c \end{bmatrix} \qquad (2.25)$$

(c) transform the coordinates to the **cylindrical** CSYS.

$$\begin{bmatrix} r \\ \varphi \\ z \end{bmatrix} = \begin{bmatrix} \sqrt{x_2^2 + y_2^2} \\ \mathrm{atan2}\,(y_2, x_2) \\ z_2 \end{bmatrix} \qquad (2.26)$$

A **cylindrical** coordinate system in FEM in the range of **small displacements** is basically a Cartesian one, just skewed for each node separately so that its **x** axis coincides with **r** axis.

Then the cylindrical to global coordinate transformation is such that:

$$
\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{global} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} r\cos\varphi \\ r\sin\varphi \\ z \end{bmatrix}_{local} + \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{origin} \tag{2.27}
$$

Therefore the Cartesian CSYS transformation matrix for definition of boundary conditions is:

(a) first subtract the cylindrical coordinate origin from the global coordinates:

$$
\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{local} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{global} - \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{origin} \tag{2.28}
$$

(b) then create a transformation matrix that maps the global csys to rotated one such that the **r** axis coincides with the **x** axis, φ axis coincides with the new **y** axis and **z** is the same:

$$
\mathbf{r}' = \frac{\mathbf{x}_{local}}{||\mathbf{x}_{local}||} \tag{2.29}
$$

$$
\mathbf{z} = \frac{\mathbf{z}_{cyl}}{||\mathbf{z}_{cyl}||} \tag{2.30}
$$

$$
\boldsymbol{\varphi} = \mathbf{z} \times \mathbf{r}' \tag{2.31}
$$

$$
\mathbf{r} = \boldsymbol{\varphi} \times \mathbf{z} \tag{2.32}
$$

the transformation matrix is then:

$$
\mathbf{T} = \begin{bmatrix} r_1 & r_2 & r_3 \\ \varphi_1 & \varphi_2 & \varphi_3 \\ z_1 & z_2 & z_3 \end{bmatrix} \tag{2.33}
$$

(c) This transformation matrix **T** is finally applied and **BC**s are imposed.

$$
\mathbf{T}^T \mathbf{K} \mathbf{T} \bar{\mathbf{u}} = \mathbf{T}^T \mathbf{f} \tag{2.34}
$$

If there already is a transformation matrix for the cylindrical system pre-
pared, such that:

$$\mathbf{x}_{local} = \mathbf{T}_{cyl}\left(\mathbf{x}_{global} - \mathbf{x}_{origin}\right) \tag{2.35}$$

or:

$$\mathbf{x}_{global} = \mathbf{T}_{cyl}^T \mathbf{x}_{local} + \mathbf{x}_{origin} \tag{2.36}$$

then:

$$r = \sqrt{x_{local}^2 + y_{local}^2} \tag{2.37}$$

$$\varphi = \text{atan2}\,\frac{y_{local}}{x_{local}} \tag{2.38}$$

$$z = z_{local} \tag{2.39}$$

because $\cos\varphi = x/r$ and $\sin\varphi = y/r$, then the new transformation matrix is
composed:

$$\mathbf{T} = \begin{bmatrix} \cos & -\sin & 0 \\ \sin & \cos & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x/r & -y/r & 0 \\ y/r & x/r & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.40}$$

and finally displacements are transformed from global to local:

$$\mathbf{u}_{local} = \mathbf{T}\mathbf{T}_{cyl}\mathbf{u}_{global} \tag{2.41}$$

Chapter 3

Multifreedom Constraints

The *multifreedom equality constraints*, or *multifreedom constraints* for short (**MFC**s) are functional equations that connect *two or more* displacement components:

$$F(nodal\ displacement\ components) = prescribed\ value \qquad (3.1)$$

where function F vanishes if all its nodal displacement arguments do. Equation (3.1) is called the *canonical form* of the constraint.

An **MFC** of this form is called a *multipoint* or *multinode* if it involves displacement components at different nodes. The constraint is called *linear* if all displacement components appear linearly on the **LHS**, and *nonlinear* otherwise.

The constraint is called *homogeneous* if, upon transfering all terms that depend on displacement components to the **LHS**, the **RHS** - the "prescribed value" in (3.1) is **zero**. It is called *nonhomogeneous* otherwise.
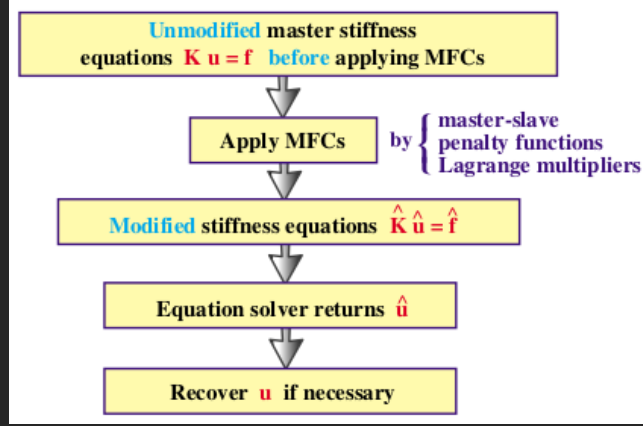
Figure 3.1: Schematics of MFC application.

3.1   Methods for Imposing MFCs

Accounting for MFCs is done, at least conceptually, by changing the assembled master stiffness equations to produce a *modified* system of equations:

$$\mathbf{K}\mathbf{u} = \mathbf{f} \stackrel{MFC}{\Longrightarrow} \hat{\mathbf{K}}\hat{\mathbf{u}} = \hat{\mathbf{f}} \tag{3.2}$$

The modification process (3.2) is also called *constraint application* or *constraint imposition*. The modified system is that submitted to the equation solver, which returns $\hat{\mathbf{u}}$.

Three methods for applying **MFCs** are written below:

(a) *Master-Slave Elimination*

The DOFs involved in each MFC are separated into master and slave freedoms. The slave freedoms are then explicitly eliminated. The modified equations do not contain the slave freedoms.

(b) *Penalty Augmentation*

Also called the *penalty function method*. Each **MFC** is viewed as the presence of a fictious elastic structural element called the *penalty element* that enforces it approximately. This element is parametrized

by a numerical *weight*. The exact constraint is recovered if the weight goes to **infinity**. The MFCs are imposed by augmenting the finite element model with the penalty elements.

(c) *Lagrange Multiplier Adjunction*

For each MFC an additional unknown is adjoined to the master stiffness equations. Physically this set of unknowns represent *constraint forces* that would enforce the constraints exactly should they be applied to the unconstrained system.

The master stiffness equations are assembled ignoring all constraints. Then the MFCs are imposed by appropriate modification of those equations. There are, however, two important practical differences:

(a) The modification process is not unique because there are alternative constraint imposition methods. These methods offer tradeofss in generality, programming implementation complexity, computational effort, numerical accuracy and stability.

(b) In the implementation of some of these methods - notably penalty augmentation - constraint imposition and assembly are carried out simultaneously. In that case the framework "first assemble, then modify", is not strictly respected in the actual implementation.

3.1.1   MFC Matrix Forms

Matrix forms of *linear* **MFCs** are often convenient for compact notation. An individual constraint may be written:

$$\begin{bmatrix} 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} u_{x2} \\ u_{x4} \\ u_{x6} \end{bmatrix} = 0.25 \tag{3.3}$$

In direct matrix notation:

$$\bar{\mathbf{a}}_i \bar{\mathbf{u}}_i = g_i, \quad (\textit{no sum on } i) \tag{3.4}$$

in which index i $(i = 1, 2, \ldots)$ identifies constraint, $\bar{\mathbf{a}}_i$ is a row vector, $\bar{\mathbf{u}}_i$ collects the set of **DOFs** that participate in the constraint, and g_i is the
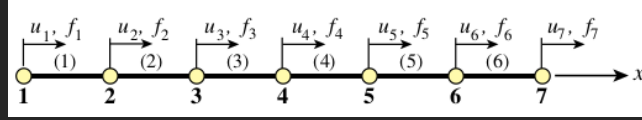
Figure 3.2: 1D problem discretized with six bar finite elements.

RHS scalar. The bars over **a** and **u** distinguishes (3.4) from the expanded form below.

For method description and general proof it is often convenient to *expand* matrix forms so that they embody *all* DOFs. For example, if (3.3) is a part of a two-dimensional FE model with 12 DOFs:

$u_{x1}, u_{y1}, \ldots, u_{y6}$, the LHS row vector may be expanded with 9 zeros as follows:

$$
\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ \vdots \\ u_{y6} \end{bmatrix} = 0.25 \qquad (3.5)
$$

In which case the matrix notation:

$$
\mathbf{a}_i \mathbf{u}_i = g_i \qquad (3.6)
$$

is used. Finally, all **MFCs** expressed as (3.6) may be collected into a single matrix relation:

$$
\mathbf{A}\mathbf{u} = \mathbf{g} \qquad (3.7)
$$

in which the rectangular matrix **A** is formed by stacking the $\mathbf{a}_i$'s as rows and column vector **g** is formed by stacking the g_is as entries. If there are 12 DOFs in **u** and **5 MFCs**, then **A** will be 5×12.

3.1.2 The Example Structure

The one-dimensional FE discretization shown in Figure 3.2 will be used to illustrate the three **MFC** application methods. The strucuture consists of six bar elements connected by seven nodes that can only displace in the x direction.

Before imposing various **multifreedom** constraints discussed below, the master stiffness equations for this problem are assumed to be:

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 \\ K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 \\ 0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 \\ 0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 \\ 0 & 0 & 0 & K_{45} & K_{55} & K_{56} & 0 \\ 0 & 0 & 0 & 0 & K_{56} & K_{66} & K_{67} \\ 0 & 0 & 0 & 0 & 0 & K_{67} & K_{77} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{bmatrix} \qquad (3.8)$$

or

$$\mathbf{Ku} = \mathbf{f} \qquad (3.9)$$

The nonzero stifness coefficients K_{ij} in (3.8) depend on the bar rigidity properties. For example, if $E^e A^e / L^e = 100$ for each element $e = 1, \ldots, 6$, then $K_{11} = K_{77} = 100$, $K_{22} = \cdots = K_{66} = 200$, $K_{12} = K_{23} = \cdots = K_{67} = -100$. However, for the purposes of the following treatment the coefficient may be kept arbitrary. The component index x in the nodal displacements u and nodal forces f has been omitted for brevity.

Now let us specify a **MFC** that states that nodes **2** and **6** must move by the same amount:

$$u_2 = u_6 \qquad (3.10)$$

Passing all node displacements to the RHS gives the canonical form:

$$u_2 - u_6 = 0 \qquad (3.11)$$

Constraint conditions of this type are sometimes called **rigid links** because they can be mechanically interpreted as forcing node points 2 and 6 to move together as if they were tied tohether by a rigid member.

We now study the imposition of constraint (3.11) on the master equations
(3.8) by the methods mentioned above. First the master-slabe method is
treated

3.2   The Master-Slave Method

To apply this method by *hand*, the MFCs are taken one at a time. For
each constraint a **slave** DOF is chosen. The freedoms remaining in that
constraint are labeled **master**. A new set of DOFs $\hat{\mathbf{u}}$ is established by
removing all slave freedoms from $\mathbf{u}$. This new vector contains master
freedoms as well as those that do not appear in the MFCs. A matrix
transformation equation that relates $\mathbf{u}$ to $\hat{\mathbf{u}}$ is generated. This equation
is used ot apply a congruent transformation to the master stiffness equa-
tions. This procedure yields a set of modified stiffness equations that are
expressed in terms of the new freedom set $\hat{\mathbf{u}}$. Because the modified system
does not contain the slave freedoms, these have been effectively eliminated.

3.2.1   A One-Constraint Example

The mechanics if the process is best seen by going through an example.
To impose (3.11) pick u_6 as slave and u_2 as master. Relate the original
unknowns $u_1, \ldots, u_7$ to the new set in which u_6 is missing:

$$
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_7 \end{bmatrix}
\tag{3.12}
$$

This is the required transformation relation. In compact form:

$$
\mathbf{u} = \mathbf{T}\hat{\mathbf{u}}
\tag{3.13}
$$

Replacing (3.12) into (3.9) and premultiplying by $\mathbf{T}^T$ yields the modified
system:

$$\hat{\mathbf{K}}\hat{\mathbf{u}} = \hat{\mathbf{f}}, \quad in \ which \quad \hat{\mathbf{K}} = \mathbf{T}^T\mathbf{K}\mathbf{T}, \quad \hat{\mathbf{f}} = \mathbf{T}^T\mathbf{f}. \qquad (3.14)$$

The form of modified system (3.14) can be remembered by a simple
mnemonic rule: premultiply both sides of $\mathbf{T}\hat{\mathbf{u}} = \mathbf{u}$ by $\mathbf{T}^T\mathbf{K}$ and replace
$\mathbf{K}\mathbf{u}$ by $\mathbf{f}$ on the right hand side.

Carrying out the indicated matrix multiplication yields:

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 & 0 \\ K_{12} & K_{22}+K_{66} & K_{23} & 0 & K_{56} & K_{67} \\ 0 & K_{23} & K_{33} & K_{34} & 0 & 0 \\ 0 & 0 & K_{34} & K_{44} & K_{45} & 0 \\ 0 & K_{56} & 0 & K_{45} & K_{55} & 0 \\ 0 & K_{67} & 0 & 0 & 0 & K_{77} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_7 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2+f_6 \\ f_3 \\ f_4 \\ f_5 \\ f_7 \end{bmatrix} \qquad (3.15)$$

Equation (3.15) is a new linear system containing 6 equations in the re-
maining 6 unknowns u_1, u_2, u_3, u_4, u_5 *and* u_7. Upon solving it, u_6 is re-
covered from the constraint (3.10).

For a simple freedom constraint such as $u_4 = 0$ the only possible choice of
slave is of course u_4 and there is no master. The congruent transformation
is then nothing more than the elimination of u_4 by striking out rows and
columns from the master stiffness equations.

3.2.2   Multiple Homogeneous MFCs

The matrix equation (3.14) in fact holds for the general case of multiple ho-
mogeneous linear constraints. Direct establishment of the transformation
equation, however, is more complicated if slave freedoms in one constraint
appear as masters in another. To illustrate this point, suppose that for
the example system we have three homogeneous MFCs:

$$u_2 - u_6 = 0$$
$$u_1 + 4u_4 = 0 \qquad (3.16)$$
$$2u_3 + u_4 + u_5 = 0$$

Picking as slave freedoms u_6, u_4 and u_3 from the first, second and third constraint, respectively, we can solve for them as:

$$u_6 = u_2$$
$$u_4 = -\frac{1}{4}u_1 \qquad (3.17)$$
$$u_3 = -\frac{1}{2}\left(u_4 + u_5\right) = \frac{1}{8}u_1 - \frac{1}{2}u_5$$

Observe that solving for u_3 from the third constraint brings u_4 to the RHS. But because u_4 is also slave freedom (it was chosen as such for the second constraint) it is replaced in favor of u_1 using $u_4 = -\frac{1}{4}u_1$. The matrix form transformation is:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \frac{1}{8} & 0 & -\frac{1}{2} & 0 \\ -\frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_5 \\ u_7 \end{bmatrix} \qquad (3.18)$$

The modified master system is now formed through the congruent transformation (3.14). Note that the slave freedoms selected from each constraint must be distinct. For example the choice u_6, u_4 and u_4 would be inadmissible as long the constraints are independent. This rule is easy to enforce when slave freedoms are chosen by hand, but can lead to implementation and numerical difficulties when it is programmed as an automated procedure, as further discussed later.

The 3 MFCs (3.16) with u_6, u_4 and u_2 chosen as slaves and u_1, u_2 and u_5 chosen as masters, may be presented in the partitioned matrix form:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 4 & 0 \\ 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} u_3 \\ u_4 \\ u_6 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_5 \end{bmatrix} \tag{3.19}$$

This may be compactly written as:

$$\mathbf{A}_s \mathbf{u}_s + \mathbf{A}_m \mathbf{u}_m = \mathbf{0} \tag{3.20}$$

Solving for the slave freedoms gives

$$\mathbf{u}_s = -\mathbf{A}_s^{-1} \mathbf{A}_m \mathbf{u}_m \tag{3.21}$$

Expanding with zeros to fill out $\mathbf{u}$ and $\hat{\mathbf{u}}$ produces (3.18). Note that non-singularity of $\mathbf{A}_s$ is essential for this method to work.

3.2.3   Nonhomoheneous MFCs

Extension to nonhomogeneous constraints is immediate. In this case the transformation equation becomes nonhomogeneous. For example suppose that (3.11) has non-zero prescribed value:

$$u_2 - u_6 = 0.2 \tag{3.22}$$

Nonzero RHS values such as 0.2 in (3.22) may often be interpreted physically as "gaps" (thus the use of the symbol $\mathbf{g}$ in the matrix form). Chose u_6 again as slave: $u_6 = u_2 - 0.2$ and build the transformation:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_7 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -0.2 \\ 0 \end{bmatrix} \tag{3.23}$$

In the compact matrix form:

$$\mathbf{u} = \mathbf{T}\hat{\mathbf{u}} + \mathbf{g} \qquad (3.24)$$

Here the constraint gap vector $\mathbf{g}$ is nonzero and $\mathbf{T}$ is the same as before. To get the modified system applying the shortcut rule, premultiply both sides of (3.24) by $\mathbf{T}^T\mathbf{K}$, replace $\mathbf{Ku}$ by $\mathbf{f}$ and pass the data to the RHS:

$$\hat{\mathbf{K}}\hat{\mathbf{u}} = \hat{\mathbf{f}}, \quad in\ which \quad \hat{\mathbf{K}} = \mathbf{T}^T\mathbf{KT}, \quad \hat{\mathbf{f}} = \mathbf{T}^T\left(\mathbf{f} - \mathbf{Kg}\right). \qquad (3.25)$$

Upon solving (3.25) for $\hat{\mathbf{u}}$, the complete displacement vector is recovered from (3.24). For the MFC (3.22) this technique gives the system:

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 & 0 \\ K_{12} & K_{22}+K_{66} & K_{23} & 0 & K_{56} & K_{67} \\ 0 & K_{23} & K_{33} & K_{34} & 0 & 0 \\ 0 & 0 & K_{34} & K_{44} & K_{45} & 0 \\ 0 & K_{56} & 0 & K_{45} & K_{55} & 0 \\ 0 & K_{67} & 0 & 0 & 0 & K_{77} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_7 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 + f_6 - 0.2K_{66} \\ f_3 \\ f_4 \\ f_5 - 0.2K_{56} \\ f_7 - 0.2K_{67} \end{bmatrix}$$
$$(3.26)$$

3.2.4   The General Case

For implementation in general-purpose programs the **master-slave method** can be described as follows. The **DOFs** in $\mathbf{u}$ are classified into three types: **independent** or unconstrained, **masters** and **slaves**. The unconstrained DOFs are those that do not appear in any MFC. Label these sets as $\mathbf{u}_u$, $\mathbf{u}_m$ and $\mathbf{u}_s$, respectively, and partition the stiffness equations accordingly:

$$\begin{bmatrix} \mathbf{K}_{uu} & \mathbf{K}_{um} & \mathbf{K}_{us} \\ \mathbf{K}_{um}^T & \mathbf{K}_{mm} & \mathbf{K}_{ms} \\ \mathbf{K}_{us}^T & \mathbf{K}_{ms}^T & \mathbf{K}_{ss} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \\ \mathbf{u}_s \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u \\ \mathbf{f}_m \\ \mathbf{f}_s \end{bmatrix} \qquad (3.27)$$

The MFCs may be written in matrix form as:

$$\mathbf{A}_m\mathbf{u}_m + \mathbf{A}_s\mathbf{u}_s = \mathbf{g}_A \qquad (3.28)$$

where $\mathbf{A}_s$ is assumed square and nonsingular. If so we can solve for the slave freedoms:

$$\mathbf{u}_s = \mathbf{A}_s^{-1}\mathbf{A}_m\mathbf{u}_m + \mathbf{A}_s^{-1}\mathbf{g}_A \triangleq \mathbf{T}\mathbf{u}_m + \mathbf{g} \tag{3.29}$$

where $\triangleq$ means *equal by definition*. Inserting into the partitioned sitffness equation (3.27) and symmetrizing yields:

$$\begin{bmatrix} \mathbf{K}_{uu} & \hat{\mathbf{K}}_{um} \\ \hat{\mathbf{K}}_{um}^T & \hat{\mathbf{K}}_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u - \mathbf{K}_{us}\mathbf{g} \\ \mathbf{f}_m - \mathbf{K}_{ms}\mathbf{g} \end{bmatrix} \tag{3.30}$$

where $\hat{\mathbf{K}}_{um} = \mathbf{K}_{um} + \mathbf{K}_{us}\mathbf{T}$
and $\hat{\mathbf{K}}_{mm} = \mathbf{K}_{mm} + \mathbf{T}^T\mathbf{K}_{ms}^T + \mathbf{K}_{ms}\mathbf{T} + \mathbf{T}^T\mathbf{K}_{ss}\mathbf{T}$.

It is seen that the misleading simplicity of the handworked example is gone.

3.2.5   Retaining the Original Freedoms

A potential disadvantage of the master-slave methid in computer work is that it requires a rearrangement of the original stiffness equations because $\hat{\mathbf{u}}$ is a subset of $\mathbf{u}$. The disadvantage can be annoying when sparse matrxi storage schemes are used for the stiffness matrix, and becomes intolerable if secondary storage is used for that purpose.

With a bit of trickery it is possible to maintain the original freedom ordering. Let us display it for the example problem under (3.11). Instead of (3.12), use the *square* transformation:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ \overline{u}_6 \\ u_7 \end{bmatrix} \tag{3.31}$$

in which $\overline{u}_6$ is a *placeholder* for the slave freedom u_6. The modified equations are:

$$
\begin{bmatrix}
K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 & 0 \\
K_{12} & K_{22}+K_{66} & K_{23} & 0 & 0 & K_{56} & 0 & K_{67} \\
0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 & 0 \\
0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 & 0 \\
0 & K_{56} & 0 & K_{45} & K_{55} & 0 & 0 & 0 \\
0 & K_{67} & 0 & 0 & 0 & 0 & 0 & K_{77}
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ \overline{u}_6 \\ u_7
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2+f_6 \\ f_3 \\ f_4 \\ f_5 \\ 0 \\ f_7
\end{bmatrix}
\tag{3.32}
$$

which are submitted to the equation solver. If the solver is not trained to skip zero rows and columns, a one should be placed in the diagonal entry for the $\overline{u}_6$ equation. The solver will return $\overline{u}_6 = 0$, and this placeholder value is replaced by u_2.

3.2.6   Model Reduction by Kinematic Constraints

The congruent transformation equations (3.14) and (3.25) have additional applications beyond the master-slave method. An important one is *model reduction by kinematic constraints*. Trough this procedure the number of **DOFs** of a static or dynamic FEM model is reduced by a significant number, typycally to $1\% - 10\%$ of the original number. This is done by taking a lot of slaves and a few masters. Only the masters are left after the transformation. The reduced model is commonly used in subsequent calculations as a component of a larger system, particularly during design or in parameter identification.

Example:

Consider the bar assembly of Figure 3.3. Assume that the only masters are the end motions u_1 and u_7, as illustrated in Figure 3.3, and interpolate all freedoms linearly:

$$
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7
\end{bmatrix}
=
\begin{bmatrix}
1 & 0 \\
\frac{5}{6} & \frac{1}{6} \\
\frac{4}{6} & \frac{2}{6} \\
\frac{3}{6} & \frac{3}{6} \\
\frac{2}{6} & \frac{4}{6} \\
\frac{1}{6} & \frac{5}{6} \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_7
\end{bmatrix},
\quad or \quad \mathbf{u} = \mathbf{T}\hat{\mathbf{u}}
\tag{3.33}
$$
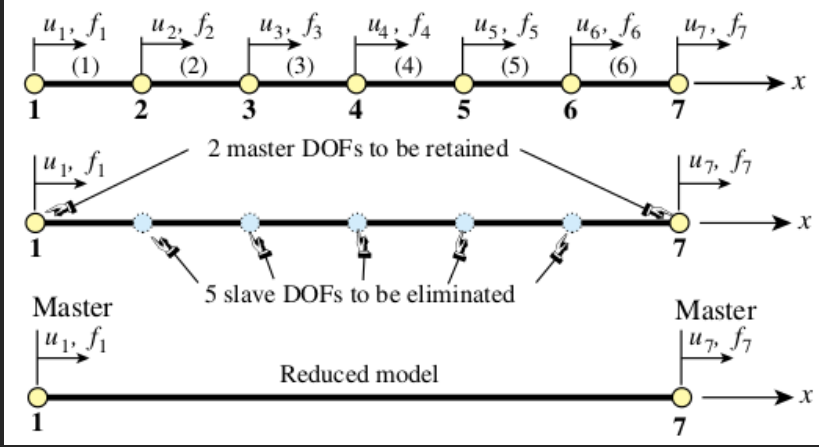
Figure 3.3: Model reduction of the example structure to the end freedoms

The reduced-order-model (**ROM**) equations are:

$$\hat{\mathbf{k}}\hat{\mathbf{u}} = \mathbf{T}^T \mathbf{K} \mathbf{T} \hat{\mathbf{u}} = \mathbf{T}^T \mathbf{f} = \hat{\mathbf{f}} \tag{3.34}$$

or in detail

$$\begin{bmatrix} \hat{K}_{11} & \hat{K}_{17} \\ \hat{K}_{17}^T & \hat{K}_{77} \end{bmatrix} \begin{bmatrix} u_1 \\ u_7 \end{bmatrix} = \begin{bmatrix} \hat{f}_1 \\ \hat{f}_7 \end{bmatrix} \tag{3.35}$$

in which:

$$\hat{K}_{11} = \frac{1}{36} \left(36K_{11} + 60K_{12} + 25K_{22} + 40K_{23} + 16K_{33} + 24K_{34} \right.$$
$$\left. + 9K_{44} + 12K_{45} + 4K_{55} + 4K_{56} + K_{66} \right)$$
$$\hat{K}_{17} = \frac{1}{36} \left(6K_{12} + 5K_{22} + 14K_{23} + 8K_{33} + 18K_{34} + 9K_{44} \right.$$
$$\left. + 18K_{45} + 8K_{55} + 14K_{56} + 5K_{66} + 6K_{67} \right)$$
$$\hat{K}_{77} = \frac{1}{36} \left(K_{22} + 4K_{23} + 4K_{33} + 12K_{34} + 9K_{44} + 24K_{45} \right. \tag{3.36}$$
$$\left. + 16K_{55} + 40K_{56} + 25K_{66} + 60K_{67} + 36K_{77} \right)$$
$$\hat{f}_1 = \frac{1}{6} \left(6f_1 + 5f_2 + 4f_3 + 3f_4 + 2f_5 + f_6 \right)$$
$$\hat{f}_7 = \frac{1}{6} \left(f_2 + 2f_3 + 3f_4 + 4f_5 + 5f_6 + 6f_7 \right)$$

This reduces the order of the FEM model from 7 to 2. The **key feature** is that the masters are picked **a priori**, as the freedoms to be retained in the model for further use.

Model reduction can be also done by the **static condensation** (Guyan) method. As its name indicates, condensation is restricted to static analysis. On the other hand, for such problems it is exact whereas model reduction by kinematic constraints generally introduces approximations.

3.2.7   Assesment of the Master-Slave Method

The **Master-Slave Method** enjoys the advantage of being **exact** (except for inevitable solution errors from finite number precision) and of reducing the nuber of unknowns. The concept is also easy to explain and learn. The main implementation drawback is the complexity of the general case. The complexity is due to three factors:

(a) The equations may have to be rearranged because of the disappearance of the slave freedoms. This drawback can be alleviated, however, by the **placeholder** trick.

(b) An auxiliary linear system has to be assembled and solved to produce the transformation matrix $\mathbf{T}$ and vector $\mathbf{g}$.

(c) The transformation process may generate many additional matrix terms. If a sparse matrix storage scheme is used for $\mathbf{K}$, the logic for allocating memory and storing these entries can be difficult and expensive.

The level of complexity depends on the generality allowed as well as on programming decisions. If $\mathbf{K}$ is stored as full matrix and slave freedom coupling in the MFCs is disallowed the logic is simple (this is the case in model reduction, since each slave freedom appears in one and only one MFC). On the other hand, if arbitrary couplings are permitted and $\mathbf{K}$ is placed in secondary (disk) storage according to some sparse scheme, the complexity can become overwhelming.

Another, more subtle, drawback of this method is that it requires decisions as to which DOFs are to be treated as slaves. This can lead to implementation and numerical stability problems. Although for disjointed constraints the process can be programmed in reliable form, in more general cases of coupled constraint equations it can lead to incorrect decisions. For example, suppose that in the example problem you have the following two MFCs:

$$
\begin{aligned}
\frac{1}{6}u_2 + \frac{1}{2}u_4 &= u_6 \\
u_3 + 6u_6 &= u_7
\end{aligned}
\tag{3.37}
$$

For numerical stability reasons it is usually better to pick as slaves the freedoms with largest coefficients. If this is done, the program would select u_6 as slave freedoms from both constraints. This leads to a contradiction because having two constraints we must eliminate two slave DOFs, not just one. The resulting modified system would in fact be inconsistent. Although this defect can be easily fixed by the program logic in this case, one can imagine the complexity burden if faced with hundreds of thousands of MFCs.

Serious numerical problems can arise if the MFCs are not independent. For example:

$$\frac{1}{6}u_2 = u_6$$
$$\frac{1}{3}u_3 + 6u_6 = u_7 \qquad (3.38)$$
$$u_2 + u_3 - u_7 = 0$$

The last constraint is an exact linear combination of the first two. If the program blindly choses u_2, u_3 and u_7 as slaves, the modified system is incorrect because we eliminate three equations when in fact there are only two independent constraints. Exact linear dependence, as before, can be recognised by a rank analysis of the $\mathbf{A}_s$ matrix defined. In the floating-point arithmetic, however, such detection may fail because that kind of computation is inexact bt nature (The safest technique to identify dependencies is to do a singular value decomposition (SVD) of $\mathbf{A}_s$. This can be, however, prohibitively expensive if one is dealing with hundreds of thousands of constraints.).

The complexithy of slave selection is in fact equivalent to that of automatically selecting kinematic redundancies in the Force Method of structural analysis. **It has led implementators of programs that use this method to require masters and slaves to be prescribed in the input data, thus transfering the burden to users**.

The method is not generally extensible to nonlinear constraints without case by case programming.

In conclusion, the master-slave method is useful when a few simple linear constraints are imposed by hand. As a general purpose technique for FEM it suffers from complexity and lack of robustness. It is worth learning, however, because of the great importance of congruent transformations in *model reduction* for static and dynamic problems.

Bibliograpy

Felippa - IFEM (Chapter 8)

Zienkiewicz and Taylor - The Finite Element Method *4th ed.* 1988
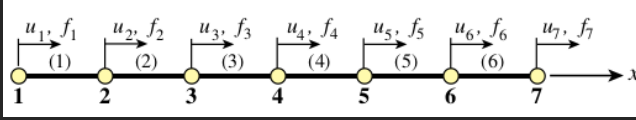
Figure 3.4: 1D problem discretized with six bar finite elements.

3.3   The Penalty Method

The penalty method will be first presented using a physical interpretation, leaving the mathematical formulation to a subsequent section. Constider again the bar example structure. To impose $u_2 = u_6$ imagine that nodes 2 and 6 are connected with a "fat" bar of axial stiffness w, labeled with element number 7, as shown in Figure 3.5. This bar is called **penalty element** and w is its **penalty weight**.

penalty element of axial rigidity w
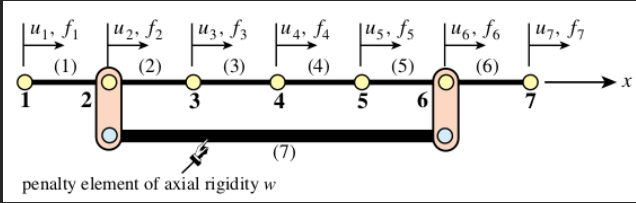
Figure 3.5: Adjusnction of a fictious penalty bar of axial stiffness w, identified as element 7, to enforce $u_2 = u_6$.

Such an element, albeit fictious, can be treated exactly like another bar element insofar as continuing the assembly of the master stiffness equations. The penalty element stiffness equations $\mathbf{K}^{(7)}\mathbf{u}^{(7)} = \mathbf{f}^{(7)}$, are:

$$w \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_2 \\ u_6 \end{bmatrix} = \begin{bmatrix} f_2^{(7)} \\ f_6^{(7)} \end{bmatrix} \tag{3.39}$$

Because there is one freedom per node, the two local element freedoms map into global freedoms 2 and 6, respectively. Using the assembly rules we obtain the following modified master stiffness equations: $\hat{\mathbf{K}}\hat{\mathbf{u}} = \hat{\mathbf{f}}$, which shown in detail are:

$$
\begin{bmatrix}
K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 \\
K_{12} & K_{22}+w & K_{23} & 0 & 0 & -w & 0 \\
0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 \\
0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 \\
0 & 0 & 0 & K_{45} & K_{55} & K_{56} & 0 \\
0 & -w & 0 & 0 & K_{56} & K_{66}+w & K_{67} \\
0 & 0 & 0 & 0 & 0 & K_{67} & K_{77}
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7
\end{bmatrix}
\tag{3.40}
$$

This system can now be submitted to the equation solver. Note that $\hat{\mathbf{u}} \equiv \mathbf{u}$ and only $\mathbf{K}$ has changed.

3.3.1   Choosing the Penalty Weight

What happens when (3.40) is solved numerically? If a *finite* weight w is chosen the constraint $u_2 = u_6$ is approximately satisfied in the sense that one gets $u_2 - u_6 = e_g$, where $e_g \neq 0$. The "gap error" e_g is called the *constraint violation*. The magnitude $|e_g|$ of this violation depends on the wight: the larger w, the smaller the violation. More precisely, it can be shown that $|e_g|$ becomes ptoportional to $\frac{1}{w}$ as w gets to be sufficiently large. For example, raising w from, say, 10^6 to 10^7 can be expected to cut the constraint violation roughly by 10 if the physical stiffnesses are smalle compared to w.

Therefore it seems as if the proper strategy should be: try to make w as large as possible while respecting the computer overflow limits. However, this is misleading. As the penalty weight w tends to ∞ the modified stiffness matrix becomes more and more *ill-conditioned with respect to inversion.*

To make this point clear, suppose for definiteness that rigidites $E^e A^e / L^e$ of the actual bars $e = 1, \ldots, 6$ are unity, that $w >> 1$, and that the computer solving the stiffness equations has a floating-point precision of 16 decimal places. Numerical analusts characterize such precision by sayinf that $\epsilon_f = \mathcal{O}(10^{-16})$, where $|\epsilon_f|$ is the smallest power of 10 that perceptibly adds to 1 in floating-point arithmetic (this is usually done not for decimal numbers but base-2 binary numbers). The modified stiffness matrix of (3.40) becomes:

$$\hat{\mathbf{K}} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2+w & -1 & 0 & 0 & -w & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & -w & 0 & 0 & -1 & 2+w & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \qquad (3.41)$$

Clearly as $w \to \infty$ rows 2 and 6, as well as columns 2 and 6, tend to become linearly dependent, in fact the negative of each other. But **linear dependency means singularity!** Thus $\hat{\mathbf{K}}$ approaches singularity as $w \to \infty$. In fact, if w exceeds $1/\epsilon_f = 10^16$ the computer will not be able to distinguish $\hat{\mathbf{K}}$ from an exactly singular matrix. If $w << 10^16$, the effect will be seen in increasing solution errors affecting the computed displacements $\hat{\mathbf{u}}$ returned by the equation solver. These errors, however, tend to be more random in nature than the constraint violation error.

3.3.2   The Square Root Rule

Obviously we have two effects at odds with each other. Making w larger reduces the constraint violation error but increases the solution error. The best w is that which makes both errors roughly equal in absolute value. This tradeoff value is difficult to find aside of systematicelly running numerical experiments. In practice *square root rule* is often followed.

This rule can be stated as follows:

Suppose that the largest stiffness coefficient, before adding penalty elements, is of the order of 10^k and that the working machine precision is p digits (such order-of magnitude estimates can be easily found by scanning just the diagonal of $\mathbf{K}$). Then choose penalty weights to be of order $10^{k+p/2}$ with the proviso that such a choice would not cause aritmetic overflow (if overflow occurs, the master stiffness matrix should be scaled throughout or a better choice of physical units made).

For the above example in which $k \approx 0$ and $p \approx 16$, the optimal w given by this rule would be $w \approx 10^8$. This w would yield a constraint violation and a solution error of order 10^{-8}. Note that there is no simple way

to do better than this accuracy aside from using **extended floating-point precision**. This is not easy to do when using standard low-level programming languages.

The name "square root@ arises because the recommended w is in fact $10^k \sqrt{10^p}$. It is seen that picking the weight by this rule requires knowledge of both stiffness magnitudes and floating-point hardware properties of the computer used, as well as the precision selected by the program.

3.3.3   Penalty Elements for General MFCs

For the constraint $u_2 = u_6$ the physical interpretation of the penalty element is clear. Points 2 and 6 must move in lockstep along x, which can be approximately enforced by the heavy bar device. But how about $3u_3 + u_5 - 4u_6 = 1$ or just $u_2 = -u_6$?

The treatment of more general constraints is linked to the theory of *Courant penalty functions* from the *variational calculus*. Consider the homogenenous constraint:

$$4u_3 + u_5 - 4u_6 = 0 \tag{3.42}$$

Rewrite this equation in matrix form:

$$\begin{bmatrix} 3 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_3 \\ u_5 \\ u_6 \end{bmatrix} = 0 \tag{3.43}$$

and premultiply both sides by the transpose of the coefficient matrix:

$$\begin{bmatrix} 3 \\ 1 \\ -4 \end{bmatrix} \begin{bmatrix} 3 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_3 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} 9 & 3 & -12 \\ 3 & 1 & -4 \\ -12 & -4 & 16 \end{bmatrix} \begin{bmatrix} u_3 \\ u_5 \\ u_6 \end{bmatrix} = \bar{\mathbf{K}}^e \mathbf{u} = 0 \tag{3.44}$$

Here $\bar{\mathbf{K}}^e$ is the *unscaleed* stiffness matrix of the penalty element. This is now multiplied by the penalty weight w and assembled into the master stiffness matrix following the usual rules. For the example problem, augmentint the master stiffness matrix with the w-scaled penalty element yields:

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 \\ K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 \\ 0 & K_{23} & K_{33}+9w & K_{34} & 3w & -12w & 0 \\ 0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 \\ 0 & 0 & 3w & K_{45} & K_{55}+w & K_{56}-4w & 0 \\ 0 & 0 & -12w & 0 & K_{56}-4w & K_{66}+16w & K_{67} \\ 0 & 0 & 0 & 0 & 0 & K_{67} & K_{77} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \end{bmatrix}$$

$$(3.45)$$

If the constraint is nonhomogeneous the force vector is also modified. To illustrate this effect, consider the MFC: $3u_3 + u_5 - 4u_6 = 1$. Rewrite this in matrix form as:

$$\begin{bmatrix} 3 & 1 & -4 \end{bmatrix} \begin{bmatrix} u_3 \\ u_5 \\ u_6 \end{bmatrix} = 1 \tag{3.46}$$

Premultiply both sides by the transpose of the coefficient matrix:

$$\begin{bmatrix} 9 & 3 & -12 \\ 3 & 1 & -4 \\ -12 & -4 & 16 \end{bmatrix} \begin{bmatrix} u_3 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ -4 \end{bmatrix} \tag{3.47}$$

Scaling by w and assembling yields:

$$\begin{bmatrix} K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 \\ K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 \\ 0 & K_{23} & K_{33}+9w & K_{34} & 3w & -12w & 0 \\ 0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 \\ 0 & 0 & 3w & K_{45} & K_{55}+w & K_{56}-4w & 0 \\ 0 & 0 & -12w & 0 & K_{56}-4w & K_{66}+16w & K_{67} \\ 0 & 0 & 0 & 0 & 0 & K_{67} & K_{77} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3+3w \\ f_4 \\ f_5+w \\ f_6-4w \\ f_7 \end{bmatrix}$$

$$(3.48)$$

3.3.4   The Theory Behind the Penalty Method Coefficient Matrix

The rule of *Courant penalty functions* comes from the following mathematical theory. Suppose we have a set of m linear MFCs. Using the

matrix notation introduced, these will be stated as:

$$\mathbf{a}_p\mathbf{u} = b_p, \quad p = 1, \ldots, m \qquad (3.49)$$

where $\mathbf{u}$ contains all DOFs and each $\mathbf{a}_p$ is a row vector with the same length as $\mathbf{u}$. To incorporate the MFCs into the FEM model one selects a weight $w_p > 0$ for each constraint and constructs the so-called **Courant quadratic penalty function** or "penalty energy":

$$P = \sum_{p=1}^{m} P_p, \; with \; P_p = \mathbf{u}^T \left(\frac{1}{2}\mathbf{a}_p^T\mathbf{a}_p\mathbf{u} - w_p\mathbf{a}_p^T b_p \right) = \frac{1}{2}\mathbf{u}^T\mathbf{K}^{(p)}\mathbf{u} - \mathbf{u}^T\mathbf{f}^{(p)}$$

$$(3.50)$$

where we have called $\mathbf{K}^{(p)} = w_p\mathbf{a}_p^T\mathbf{a}_p$ and $\mathbf{f}^{(p)} = w_p\mathbf{a}^T b_p$. P is added to the potential energy function $\prod = \frac{1}{2}\mathbf{u}^T\mathbf{K}\mathbf{u} - \mathbf{u}^T\mathbf{f}$ to form the *augmented potential energy* $\prod_a = \prod + P$. Minimization of $\prod_a$ with respect to $\mathbf{u}$ yields:

$$\left(\mathbf{K}\mathbf{u} + \sum_{p=1}^{m} \mathbf{K}^{(p)} \right) \mathbf{u} = \mathbf{f} + \sum_{p=1}^{m} \mathbf{f}^{(p)} \qquad (3.51)$$

Each term of the sum on p, which derives from term P_p may be viewed as contributed by a penalty element with globalised stiffness matrix $\mathbf{K}^{(p)} = w_p\mathbf{a}_p^T\mathbf{a}_p$ and globalised added force term $\mathbf{f}^{(p)} = w_p\mathbf{a}_p^T b_p$.

To use a even more compact form we may write the set of MFCs as $\mathbf{A}\mathbf{u} = \mathbf{b}$. Then the penalty augmented system can be written compactly as:

$$\left(\mathbf{K} + \mathbf{A}^T\mathbf{W}\mathbf{A} \right) \mathbf{u} = \mathbf{f} + \mathbf{W}\mathbf{A}^T\mathbf{b} \qquad (3.52)$$

where $\mathbf{W}$ is a diagonal matrix of penalty weights. This compact form, however, conceals the configuration of the penalty elements.

3.3.5   Assesment of the Penalty Method

The main advantage of the penalty function method is its straightforward computer implementation. Looking at the modified systems it is

obvious that master equations need not be rearranged. That is, **u** and **û** are the same. Constraints may be programmed as "penalty elements", and stiffness and force contributions fo these elements merged through the standard assembler. In fact using this method there is no need to distinguish between unconstrained and constrained equations! Once all elements - regular and penalty - are assembled, the system can be passed to the equation solver. **Single Freedom Constraints**, are usually processed separately for efficiency.

An important advantage with respect to the master-slave (elimination) method is its lack of sensitivity with respect to whether constraints are linearly dependent. To give a simplistic example, suppose that the constraint $u_2 = u_6$ appears twice. Then two penalty elements connecting 2 and 6 will be inserted, doubling the intended weight but not otherwise causing undue harm.

An advantage with respect to the Lagrange multiplier method is that positive definiteness is not lost. Such loss can affect the performance of certain numerical processes (for example solving the master stiffness equations by Cholesky factorisation or conjugate-gradients). Finally, it is worth noting that the penalty method is easily extensible to nonlinear constraints.

The main disadvantage, however, is a serious one: the choice of weight values that balance solution accuracy with the violation of constraint conditions. For simple cases the square root rule previously described often works, although its effective use calls for knowledge og the magnitude of stiffness coefficients. Such knowledge may be difficult to extract from general purpose "black box" program. For difficult cases selection of appropriate weights may require extensive numerical experimentation, wasting the user time with numerical games that have no bearing on the actual objective, which is getting a solution.

The deterioration of the condition number fo the penalty-augmented stiffness matrix can have serious side effects in some solution procedures such as eigenvalue extraction or iterative solver.

Finally, even if optimal weights are selected, the combined solution error cannot be lowered beyond a treshold value.

From this assessment it is evident that penalty augmentation, although superior to the master-slave method from the standpoint of generality and ease of implementation, is no panacea.

3.4   Lagrange Multiplier Adjuction
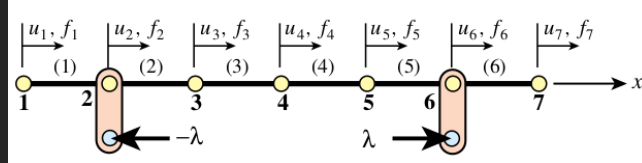
Figure 3.6: Physical interpretation of Lagrange multiplier adjuction to enforce the MFC $u_2 = u_6$

3.4.1   Physical Interpretation

As in the case of the penalty function method, the method Lagrange multiplier can be given a rigorous justification within the framework of variational calculus. But in the same spirit it will be introduced for the example structure from a physical standpoint that is particularly illuminating.

Consider again the constraint $u_2 = u_6$. Borrowing some ideas from the penalty method, imagine that nodes 2 and 6 are connected now by a *rigis* link rather than a flexible one. Thus the constraint is imposed exactly. But of course the penalty method with an infinite weight would "blow up".

We may remove the link if it is replaced by an appropriate reaction force pair $(-\lambda, +\lambda)$, as illustrated in Figure 3.6. These are called the *constraint forces*. Incorporating these forces into the original stiffness equations we get:

$$
\begin{bmatrix}
K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 \\
K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 \\
0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 \\
0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 \\
0 & 0 & 0 & K_{45} & K_{55} & K_{56} & 0 \\
0 & 0 & 0 & 0 & K_{56} & K_{66} & K_{67} \\
0 & 0 & 0 & 0 & 0 & K_{67} & K_{77}
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2 - \lambda \\ f_3 \\ f_4 \\ f_5 \\ f_6 + \lambda \\ f_7
\end{bmatrix}
\tag{3.53}
$$

This λ is called **Lagrange multiplier**. Because λ is an unknown, let us transfer it to the **LHS** by **appending** it to the vector of unknowns:

$$
\begin{bmatrix}
K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 & 0 \\
K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 & 1 \\
0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 & 0 \\
0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 & 0 \\
0 & 0 & 0 & K_{45} & K_{55} & K_{56} & 0 & 0 \\
0 & 0 & 0 & 0 & K_{56} & K_{66} & K_{67} & -1 \\
0 & 0 & 0 & 0 & 0 & K_{67} & K_{77} & 0
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ \lambda
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7
\end{bmatrix}
\quad (3.54)
$$

But now we have 7 equations for 8 unknowns. To render the system determinate, the constraint condition $u_2 - u_6 = 0$ is appended as eight equation:

$$
\begin{bmatrix}
K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 & 0 \\
K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 & 1 \\
0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 & 0 \\
0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 & 0 \\
0 & 0 & 0 & K_{45} & K_{55} & K_{56} & 0 & 0 \\
0 & 0 & 0 & 0 & K_{56} & K_{66} & K_{67} & -1 \\
0 & 0 & 0 & 0 & 0 & K_{67} & K_{77} & 0 \\
0 & 1 & 0 & 0 & 0-1 & 0 & 0 &
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ \lambda
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ 0
\end{bmatrix}
\quad (3.55)
$$

This is called the **multiplier-augmented** system. Its coefficient matrix, which is symmetric, is called the **bordered stiffness matrix**. The process by which λ is appended to the vector of original unknowns is called **adjucation**. Solving this system provides the desired solution for the DOFs while also characterising the constraint forces through λ.

3.4.2 Lagrange Multipliers fro General MFCs

The general procedure will be stated first as a recipe. Suppose that we want to solve the example structure subjected to three MFCs:

$$
u_2 - u_6 = 0, \quad 5u_2 - 8u_7 = 3, \quad 3u_3 + u_5 - 4u_6 = 1 \quad (3.56)
$$

Adjoin these MFCs as the eighth, nineth, and tenth equations:

$$
\begin{bmatrix}
K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 \\
K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 \\
0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 \\
0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 \\
0 & 0 & 0 & K_{45} & K_{55} & K_{56} & 0 \\
0 & 0 & 0 & 0 & K_{56} & K_{66} & K_{67} \\
0 & 0 & 0 & 0 & 0 & K_{67} & K_{77} \\
0 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 5 & 0 & 0 & 0 & 0 & -8 \\
0 & 0 & 3 & 0 & 1 & -4 & 0
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ 0 \\ 3 \\ 1
\end{bmatrix}
\tag{3.57}
$$

Three Lagrange multipliers: λ_1, λ_2 and λ_3 are required to care of three MFCs. Adjoin those unknowns to the nodal displacement vector. Symmetrize the coefficient matrix by appending 3 columns that are the transpose of the 3 last rows and filling the bottom righ-hand corner with a 3×3 zero matrix:

$$
\begin{bmatrix}
K_{11} & K_{12} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
K_{12} & K_{22} & K_{23} & 0 & 0 & 0 & 0 & 1 & 5 & 0 \\
0 & K_{23} & K_{33} & K_{34} & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 0 & K_{34} & K_{44} & K_{45} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & K_{45} & K_{55} & K_{56} & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & K_{56} & K_{66} & K_{67} & -1 & 0 & -4 \\
0 & 0 & 0 & 0 & 0 & K_{67} & K_{77} & 0 & -8 & 0 \\
0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 5 & 0 & 0 & 0 & 0 & -8 & 0 & 0 & 0 \\
0 & 0 & 3 & 0 & 1 & -4 & 0 & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ \lambda_1 \\ \lambda_2 \\ \lambda_3
\end{bmatrix}
=
\begin{bmatrix}
f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ 0 \\ 3 \\ 1
\end{bmatrix}
\tag{3.58}
$$

3.4.3   The Theory Behind

The recipe illustrated above comes from a well known technique of variational calculus. Using the matrix notation introduced, compactly denote the set of m MFCs by $\mathbf{A}\mathbf{u} = \mathbf{b}$, where $\mathbf{A}$ is $m \times n$. The potential energy of the unconstrained finite element model is $\prod = \frac{1}{2}\mathbf{u}^T\mathbf{K}\mathbf{u} - \mathbf{u}^T\mathbf{f}$. To impose

the constraint, adjoin m Lagrange multipliers collected in vector λ and form the Lagrangian

$$\mathcal{L}(\mathbf{u},\boldsymbol{\lambda}) = \prod +\boldsymbol{\lambda}\left(\mathbf{Au} - \mathbf{b}\right) = \frac{1}{2}\mathbf{u}^T\mathbf{Ku} - \mathbf{u}^T\mathbf{f} + \boldsymbol{\lambda}^T\left(\mathbf{Au} - \mathbf{b}\right) \qquad (3.59)$$

Extremizing $\mathcal{L}$ with respect to $\mathbf{u}$ and $\boldsymbol{\lambda}$ yields the multiplier-augmented form

$$\begin{bmatrix} \mathbf{K} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \mathbf{b} \end{bmatrix} \qquad (3.60)$$

The master stiffness matrix $\mathbf{K}$ is said to be **bordered** with $\mathbf{A}$ and $\mathbf{A}^T$. Solving this system provides $\mathbf{u}$ and $\boldsymbol{\lambda}$. The latter can be interpreted as forces of constraint in the following sense: a removed constraint can be replaced by a system of forces characterised by $\boldsymbol{\lambda}$ multiplied by the constraint coefficients. More precisely, the constraint forces are $-\mathbf{A}^T\boldsymbol{\lambda}$.

3.4.4 Assesment of the Lagrange Multiplier Method

In contrast to the penalty method, the method of Lagrange multipliers has the advantage of being exact (aside from computational errors due to finite precision arithmetic). It provides directly the constraint forces, which are of interest in many applications. It does not require guesses as regards to weights. As the penalty method, it can be extended without difficulty to nonlinear constraints.

It is not free of disadvantages. It introduces additional unknowns, requiring expansion of the original stiffness method, and more complicated storage allocation procedures. It renders the augmented stiffness matrix indefinite, na effect that may cause grief with some linear equation solving methods that rely on positive definiteness. Finally, as the master-slave method, it is sensitive to the degree of linear independence of the constraints: if the constraint $u_2 = u_6$ is specified twice, the bordered stiffness is obviously singular.

On the whole this method appears to be the most elegant one for a general-purpose finite element program that is supposed to work as a "black box" by minimizing guesses and choices from its users. Its implementation,

however, is not simple. Special care must be excercised to detect singularities due to constraint dependency and to account for the effect of loss of positive definiteness of the bordered stiffness on equation solver.

3.4.5 The Augmented Lagrangian Method

The general matrix of the penalty function and Lagrangian multiplier methods are given by expressions (3.51) and (3.60), respectively. A useful connection between these methods can be established as follows.

Because the lowe diagonal block of the bordered stiffness matrix in (3.60) is null, it is not possible to directly eliminate $\boldsymbol{\lambda}$. To make this possible, replace this block by $\epsilon \mathbf{S}^{-1}$, where $\mathbf{S}$ is a constraint-scaling diagonal matrix of appropriate order and ϵ is a small number. The reciprocal of ϵ is a large number called $w = 1/\epsilon$. To maintain exactness of the second equation, $\epsilon \mathbf{S}^{-1} \boldsymbol{\lambda}$ is added to the RHS:

$$\begin{bmatrix} \mathbf{K} & \mathbf{A}^T \\ \mathbf{A} & \epsilon \mathbf{S}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ \epsilon \mathbf{S}^{-1} \boldsymbol{\lambda}^P \end{bmatrix} \qquad (3.61)$$

Here superscript P (for "predicted" value) is attached to the $\boldsymbol{\lambda}$ on the RHS as a "tracer". We can now formally solve for $\boldsymbol{\lambda}$ and subsequently for $\mathbf{u}$. The results may be presented as:

$$\left(\mathbf{K} + w \mathbf{A}^T \mathbf{S} \mathbf{A} \right) \mathbf{u} = \mathbf{f} + w \mathbf{A}^T \mathbf{S} \mathbf{b} - \mathbf{A}^T \boldsymbol{\lambda}^P$$
$$\boldsymbol{\lambda} = \boldsymbol{\lambda}^P + w \mathbf{S} \left(\mathbf{b} - \mathbf{A} \mathbf{u} \right) \qquad (3.62)$$

Setting $\boldsymbol{\lambda}^P = \mathbf{0}$ in the first matrix equation yields:

$$\left(\mathbf{K} + w \mathbf{A}^T \mathbf{S} \mathbf{A} \right) \mathbf{u} = \mathbf{f} + w \mathbf{A}^T \mathbf{S} \mathbf{b} \qquad (3.63)$$

on taking $\mathbf{W} = w\mathbf{S}$, the general matrix equation (3.51) of the penalty method is recovered.

This relation suggests the construction of *iterative procedures* in which one tries to *improve the accuracy of the penalty function while w is kept constant*. This strategy circumvents the aforementioned ill-conditioning problems when the weight w is gradually increased. One such method is

easily constructed by inspecting (3.62). Using superscript k as an iteration index and keeping w fixed, solve equations (3.62) in tandem as follows:

$$\left(\mathbf{K} + \mathbf{A}^T \mathbf{W} \mathbf{A}\right) \mathbf{u}^k = \mathbf{f} + \mathbf{A}^T \mathbf{W} \mathbf{b} - \mathbf{A}^T \boldsymbol{\lambda}^k$$
$$\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + \mathbf{W}\left(\mathbf{b} - \mathbf{A}\mathbf{u}^k\right) \tag{3.64}$$

for $k = 0, 1, \ldots$, beginning with $\boldsymbol{\lambda}^0 = 0$. Then $\mathbf{u}^0$ is the penalty solution. If the process converges one recovers the exact Lagrangian solution without having to solve the Lagrangian system (3.61) directly.

The family of iterative procedures that may be precipitated from (3.62) collectively pertains to the class of *augmented Lagrangian methods*.

3.5 Summary

The treatment of linear MFCs in finite element systems can be carried out by several methods. Three of these: **master-slave elimination**, **penalty augmentation** and **Lagrange multiplier** adjunction have been discussed.

It is emphasised that no method is uniformly satisfactory in terms of generality, robustness, numerical behavior and simplicity of implementation.

For a general purpose program that tries to attain "black box" behavior (that is minimal decisions on the part of users) the Lagrange multipliers has the edge. This edge is unfortunately blunted by a fairly complex computer implementation and by the loss of positive definiteness in the bordered stiffness matrix.

Chapter 4

Rigid Body Elements

4.1 RBE2

The rigid freebody relations between **slave** and **master** node are:

$$\mathbf{T}_s = \mathbf{T}_m + \mathbf{R}_s \times \overline{\mathbf{x}}$$
$$\mathbf{R}_s = \mathbf{R}_m$$

(4.1)

Where $\overline{\mathbf{x}}$ is a vector from **slave** to **master**:

$$\overline{\mathbf{x}} = \begin{bmatrix} \overline{x} \\ \overline{y} \\ \overline{z} \end{bmatrix} = \begin{bmatrix} x_s - x_m \\ y_s - y_m \\ z_s - z_m \end{bmatrix}$$

(4.2)

Then *vector multiplication* (cross product) is defined as:

$$
\begin{aligned}
\mathbf{R}_s \times \mathbf{x}_s = \mathbf{R}_m \times \mathbf{x} = & \\
= \begin{bmatrix} \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix} \times \begin{bmatrix} x_s - x_m \\ y_s - y_m \\ z_s - z_m \end{bmatrix} \quad\quad &= \begin{bmatrix} \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix} \times \begin{bmatrix} \overline{x} \\ \overline{y} \\ \overline{z} \end{bmatrix} = \\
= \begin{bmatrix} \psi_m(z_s - z_m) - \theta_m(y_s - y_m) \\ \theta_m(x_s - x_m) - \varphi_m(z_s - z_m) \\ \varphi_m(y_s - y_m) - \psi_m(x_s - x_m) \end{bmatrix} &= \begin{bmatrix} \psi_m\overline{z} - \theta_m\overline{y} \\ \theta_m\overline{x} - \varphi_m\overline{z} \\ \varphi_m\overline{y} - \psi_m\overline{x} \end{bmatrix} = \\
= \begin{bmatrix} 0 & \overline{z} & -\overline{y} \\ -\overline{z} & 0 & \overline{x} \\ \overline{y} & -\overline{x} & 0 \end{bmatrix} \begin{bmatrix} \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix} &
\end{aligned}
$$

(4.3)

The **slave** equations can be written as:

$$
\begin{bmatrix} u_s \\ v_s \\ w_s \\ \varphi_s \\ \psi_s \\ \theta_s \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & z_s - z_m & -(y_s - y_m) \\ 0 & 1 & 0 & -(z_s - z_m) & 0 & x_s - x_m \\ 0 & 0 & 1 & y_s - y_m & -(x_s - x_m) & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_m \\ v_m \\ w_m \\ \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix} \quad (4.4)
$$

or:

$$
\begin{bmatrix} u_s \\ v_s \\ w_s \\ \varphi_s \\ \psi_s \\ \theta_s \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & \overline{z} & -\overline{y} \\ 0 & 1 & 0 & -\overline{z} & 0 & \overline{x} \\ 0 & 0 & 1 & \overline{y} & -\overline{x} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_m \\ v_m \\ w_m \\ \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix} \quad (4.5)
$$

4.1.1   RBE2 Coefficients Example:

Master Node:

$$\mathbf{X}_m = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T \tag{4.6}$$

Slave Nodes:

$$
\begin{aligned}
\mathbf{X}_{s,1} &= \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \\
\mathbf{X}_{s,1} &= \begin{bmatrix} 2 & 0 & 0 \end{bmatrix}^T \\
\mathbf{X}_{s,1} &= \begin{bmatrix} 2 & 2 & 0 \end{bmatrix}^T \\
\mathbf{X}_{s,1} &= \begin{bmatrix} 0 & 2 & 0 \end{bmatrix}^T
\end{aligned} \tag{4.7}
$$

Then Slave Node 1 equations are:

$$\mathbf{x}_1 = \begin{bmatrix} x_{s,1} - x_m & y_{s,1} - y_m & z_{s,1} - z_m \end{bmatrix}^T = \begin{bmatrix} -1 & -1 & -1 \end{bmatrix}^T \tag{4.8}$$

Other Slave Nodes:

$$
\begin{aligned}
\mathbf{x}_2 &= \begin{bmatrix} 2-1 & 0-1 & 0-1 \end{bmatrix}^T = \begin{bmatrix} 1 & -1 & -1 \end{bmatrix}^T \\
\mathbf{x}_3 &= \begin{bmatrix} 2-1 & 2-1 & 0-1 \end{bmatrix}^T = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix}^T \\
\mathbf{x}_4 &= \begin{bmatrix} 0-1 & 2-1 & 0-1 \end{bmatrix}^T = \begin{bmatrix} -1 & 1 & -1 \end{bmatrix}^T
\end{aligned} \tag{4.9}
$$

Therefore, after filling in the coefficients:

$$
\begin{bmatrix} u_{s,1} \\ v_{s,1} \\ w_{s,1} \\ \varphi_{s,1} \\ \psi_{s,1} \\ \theta_{s,1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_m \\ v_m \\ w_m \\ \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix}
$$

$$
\begin{bmatrix} u_{s,2} \\ v_{s,2} \\ w_{s,2} \\ \varphi_{s,2} \\ \psi_{s,2} \\ \theta_{s,2} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_m \\ v_m \\ w_m \\ \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix}
$$

$$
\begin{bmatrix} u_{s,3} \\ v_{s,3} \\ w_{s,3} \\ \varphi_{s,3} \\ \psi_{s,3} \\ \theta_{s,3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_m \\ v_m \\ w_m \\ \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix}
$$

$$
\begin{bmatrix} u_{s,4} \\ v_{s,4} \\ w_{s,4} \\ \varphi_{s,4} \\ \psi_{s,4} \\ \theta_{s,4} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_m \\ v_m \\ w_m \\ \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix}
$$

(4.10)

When all put together:

$$
\begin{bmatrix} u_{s,1} \\ v_{s,1} \\ w_{s,1} \\ \varphi_{s,1} \\ \psi_{s,1} \\ \theta_{s,1} \\ u_{s,2} \\ v_{s,2} \\ w_{s,2} \\ \varphi_{s,2} \\ \psi_{s,2} \\ \theta_{s,2} \\ u_{s,3} \\ v_{s,3} \\ w_{s,3} \\ \varphi_{s,3} \\ \psi_{s,3} \\ \theta_{s,3} \\ u_{s,4} \\ v_{s,4} \\ w_{s,4} \\ \varphi_{s,4} \\ \psi_{s,4} \\ \theta_{s,4} \end{bmatrix}
=
\begin{bmatrix}
1 & 0 & 0 & 0 & -1 & 1 \\
0 & 1 & 0 & 1 & 0 & -1 \\
0 & 0 & 1 & -1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & -1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & -1 & -1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & -1 & -1 \\
0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & -1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & -1 & -1 \\
0 & 1 & 0 & 1 & 0 & -1 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix} u_m \\ v_m \\ w_m \\ \varphi_m \\ \psi_m \\ \theta_m \end{bmatrix}
\qquad (4.11)
$$

It can be read as:

$$
\begin{aligned}
u_{s,1} &= u_m - \psi_m + \theta_m \\
v_{s,1} &= v_m \varphi_m - \theta_m \\
&\;\;\vdots \\
\theta_{s,4} &= \theta_m
\end{aligned}
\qquad (4.12)
$$

Or:

$$
\mathbf{u}_s = \bar{\mathbf{T}} \mathbf{u}_m \qquad (4.13)
$$

4.1.2   Editing the Master Stiffness Matrix

This equation should be then expanded to all DOFs, where in $\mathbf{T}$ the slave DOFs columns are missing and the DOFs that do not take part in RBEs have 1 on the diagonal. The resulting matrix should have the same number of rows as the master stiffness matrix.

Then the master stiffness equation is modified:

$$\hat{\mathbf{K}} = \mathbf{T}^T \mathbf{K} \mathbf{T}$$
$$\hat{\mathbf{f}} = \mathbf{T}^T \mathbf{f} \tag{4.14}$$
$$\hat{\mathbf{K}}\hat{\mathbf{u}} = \hat{\mathbf{f}}$$

If we'd like to make the procedure more general, partition the master stiffness equations such that:

$$
\begin{bmatrix}
\mathbf{K}_{uu} & \mathbf{K}_{um} & \mathbf{K}_{us} \\
\mathbf{K}_{um}^T & \mathbf{K}_{mm} & \mathbf{K}_{ms} \\
\mathbf{K}_{us}^T & \mathbf{K}_{ms}^T & \mathbf{K}_{ss}
\end{bmatrix}
\begin{bmatrix}
\mathbf{u}_u \\
\mathbf{u}_m \\
\mathbf{u}_s
\end{bmatrix}
=
\begin{bmatrix}
\mathbf{f}_u \\
\mathbf{f}_m \\
\mathbf{f}_s
\end{bmatrix}
\tag{4.15}
$$

Based on (4.14) we could probably partition the matrices as:

$$
\begin{array}{c}
\quad\; u \quad m \\
\begin{array}{c} u \\ m \\ s \end{array}
\left|
\begin{bmatrix}
\mathbf{I} & \mathbf{0} \\
\mathbf{0} & \mathbf{I} \\
\mathbf{0} & \overline{\mathbf{T}}
\end{bmatrix}
\right.
\end{array}
\tag{4.16}
$$

If we expand (4.13) and partition the matrices based on (4.16) into independent or **unconstrained**, **master** and **slave** nodes, then the equation:

$$\mathbf{u} = \mathbf{T}\hat{\mathbf{u}} \tag{4.17}$$

becomes:

$$
\begin{bmatrix}
\mathbf{u}_u \\
\mathbf{u}_m \\
\mathbf{u}_s
\end{bmatrix}
=
\begin{bmatrix}
\mathbf{I} & \mathbf{0} \\
\mathbf{0} & \mathbf{I} \\
\mathbf{0} & \overline{\mathbf{T}}
\end{bmatrix}
\begin{bmatrix}
\mathbf{u}_u \\
\mathbf{u}_m
\end{bmatrix}
\tag{4.18}
$$

Finally the equation (4.14) can be rewritten as:

$$
\begin{bmatrix}
\mathbf{I} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{I} & \overline{\mathbf{T}}^T
\end{bmatrix}
\begin{bmatrix}
\mathbf{K}_{uu} & \mathbf{K}_{um} & \mathbf{K}_{us} \\
\mathbf{K}_{mu} & \mathbf{K}_{mm} & \mathbf{K}_{ms} \\
\mathbf{K}_{su} & \mathbf{K}_{sm} & \mathbf{K}_{ss}
\end{bmatrix}
\begin{bmatrix}
\mathbf{I} & \mathbf{0} \\
\mathbf{0} & \mathbf{I} \\
\mathbf{0} & \overline{\mathbf{T}}
\end{bmatrix}
\begin{bmatrix}
\mathbf{u}_u \\
\mathbf{u}_m
\end{bmatrix}
=
\begin{bmatrix}
\mathbf{I} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{I} & \overline{\mathbf{T}}^T
\end{bmatrix}
\begin{bmatrix}
\mathbf{f}_u \\
\mathbf{f}_m \\
\mathbf{f}_s
\end{bmatrix}
\tag{4.19}
$$

Expanding and multiplying:

$$\begin{bmatrix} \mathbf{K}_{uu} & \mathbf{K}_{um} & \mathbf{K}_{us} \\ \mathbf{K}_{mu} + \overline{\mathbf{T}}^T \mathbf{K}_{su} & \mathbf{K}_{mm} + \overline{\mathbf{T}}^T \mathbf{K}_{sm} & \mathbf{K}_{ms} + \overline{\mathbf{T}}^T \mathbf{K}_{ss} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \overline{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u \\ \mathbf{f}_m + \overline{\mathbf{T}}^T \mathbf{f}_s \end{bmatrix}$$
$$(4.20)$$

$$\begin{bmatrix} \mathbf{K}_{uu} & \mathbf{K}_{um} + \mathbf{K}_{us}\overline{\mathbf{T}} \\ \mathbf{K}_{mu} + \overline{\mathbf{T}}^T \mathbf{K}_{su} & \mathbf{K}_{mm} + \overline{\mathbf{T}}^T \mathbf{K}_{sm} + \mathbf{K}_{ms}\overline{\mathbf{T}} + \overline{\mathbf{T}}^T \mathbf{K}_{ss}\overline{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u \\ \mathbf{f}_m + \overline{\mathbf{T}}^T \mathbf{f}_s \end{bmatrix}$$
$$(4.21)$$

which is equal to:

$$\begin{bmatrix} \mathbf{K}_{uu} & \mathbf{K}_{um} + \mathbf{K}_{us}\overline{\mathbf{T}} \\ (\mathbf{K}_{um} + \mathbf{K}_{us}\overline{\mathbf{T}})^T & \mathbf{K}_{mm} + \overline{\mathbf{T}}^T \mathbf{K}_{ms}^T + \mathbf{K}_{ms}\overline{\mathbf{T}} + \overline{\mathbf{T}}^T \mathbf{K}_{ss}\overline{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u \\ \mathbf{f}_m + \overline{\mathbf{T}}^T \mathbf{f}_s \end{bmatrix}$$
$$(4.22)$$

considering that $\mathbf{f}_s = \mathbf{0}$ the equations reduce to:

$$\begin{bmatrix} \mathbf{K}_{uu} & \hat{\mathbf{K}}_{um} \\ \hat{\mathbf{K}}_{um}^T & \hat{\mathbf{K}}_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u \\ \mathbf{f}_m \end{bmatrix} \qquad (4.23)$$

where:

$$\hat{\mathbf{K}}_{um} = \mathbf{K}_{um} + \mathbf{K}_{us}\overline{\mathbf{T}}$$
$$\hat{\mathbf{K}}_{mm} = \mathbf{K}_{mm} + \overline{\mathbf{T}}^T \mathbf{K}_{ms}^T + \mathbf{K}_{ms}\overline{\mathbf{T}} + \overline{\mathbf{T}}^T \mathbf{K}_{ss}\overline{\mathbf{T}} \qquad (4.24)$$
$$= \mathbf{K}_{mm} + \mathbf{K}_{ms}\overline{\mathbf{T}} + (\mathbf{K}_{ms}\overline{\mathbf{T}})^T + \overline{\mathbf{T}}^T \mathbf{K}_{ss}\overline{\mathbf{T}}$$

Then if **master-slave** relations are written as (*general case*):

$$\mathbf{u}_s = \bar{\mathbf{T}}\mathbf{u}_m + \mathbf{g} \tag{4.25}$$

Inserting into the partitioned master stiffness equations:

$$\begin{bmatrix} \mathbf{K}_{uu} & \hat{\mathbf{K}}_{um} \\ \hat{\mathbf{K}}_{um}^T & \hat{\mathbf{K}}_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u - \mathbf{K}_{us}\mathbf{g} \\ \mathbf{f}_m - \mathbf{K}_{ms}\mathbf{g} \end{bmatrix} \tag{4.26}$$

where:

$$\begin{aligned} \hat{\mathbf{K}}_{um} &= \mathbf{K}_{um} + \mathbf{K}_{us}\bar{\mathbf{T}} \\ \hat{\mathbf{K}}_{mm} &= \mathbf{K}_{mm} + \bar{\mathbf{T}}^T\mathbf{K}_{ms}^T + \mathbf{K}_{ms}\bar{\mathbf{T}} + \bar{\mathbf{T}}^T\mathbf{K}_{ss}\bar{\mathbf{T}} \end{aligned} \tag{4.27}$$

MPC Application

Afterwards the **MPCs** are applied:

$$\mathbf{u}_s = \bar{\mathbf{T}}\mathbf{u}_m \tag{4.28}$$

This, when applied to the whole model:

$$\begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \\ \mathbf{u}_s \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \bar{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} \tag{4.29}$$

with **transformation matrix T** being:

$$\mathbf{T} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \bar{\mathbf{T}} \end{bmatrix} \tag{4.30}$$

Applying the transformation matrix to get **modified master equation system**:

$$\begin{aligned} \mathbf{T}^T\mathbf{K}_g\mathbf{T}\hat{\mathbf{u}}_g &= \mathbf{T}^T\mathbf{f}_g \\ \hat{\mathbf{K}}_g\hat{\mathbf{u}}_g &= \hat{\mathbf{f}}_g \end{aligned} \tag{4.31}$$

Where $\hat{\mathbf{u}}_g = \begin{bmatrix} \mathbf{u}_{g,u} & \mathbf{u}_{g,m} \end{bmatrix}^T$.

From now on, the subscript $_g$ (*as global*) is implied.

Rotated Nodal Basis

When a node or an SPC is defined as a **rotated/skewed** or in a different type of Coordinate system (e.g. cylindrical) a transformation matrix needs to be applied to such nodes.

Cartesian Coordinate system definition by angle (2D):

$$\mathbf{u}_l = \mathbf{T}\mathbf{u}_g \tag{4.32}$$

SPC Application

Then to solve the **master stiffness equations** for prescribed **BCs** partitioned to **unconstrained** and **master** DOFs such as:

$$\begin{bmatrix} \mathbf{K}_{uu} & \hat{\mathbf{K}}_{um} \\ \hat{\mathbf{K}}_{um}^T & \hat{\mathbf{K}}_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} = \begin{bmatrix} \mathbf{f}_u \\ \mathbf{f}_m \end{bmatrix} \tag{4.33}$$

where:

$$\begin{aligned} \hat{\mathbf{K}}_{um} &= \mathbf{K}_{um} + \mathbf{K}_{us}\overline{\mathbf{T}} \\ \hat{\mathbf{K}}_{mm} &= \mathbf{K}_{mm} + \mathbf{K}_{ms}\overline{\mathbf{T}} + (\mathbf{K}_{ms}\overline{\mathbf{T}})^T + \overline{\mathbf{T}}^T\mathbf{K}_{ss}\overline{\mathbf{T}} \end{aligned} \tag{4.34}$$

Any of these **DOFs** can be constrained, so this partitioning scheme looses all purposes. Therefore the matrix is repartitioned again into **independent** and **constrained** partitions:

$$\begin{bmatrix} \mathbf{K}_{ii} & \mathbf{K}_{ic} \\ \mathbf{K}_{ic}^T & \mathbf{K}_{cc} \end{bmatrix} \begin{bmatrix} \mathbf{u}_i \\ \mathbf{u}_c \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i \\ \mathbf{f}_c \end{bmatrix} \tag{4.35}$$

Solving Linear Equations

Where the **unknowns** are $\mathbf{u}_i$, $\mathbf{f}_c$:

$$\begin{bmatrix} \mathbf{K}_{ii} & \mathbf{K}_{ic} \\ \mathbf{K}_{ic}^T & \mathbf{K}_{cc} \end{bmatrix} \begin{bmatrix} \boxed{\mathbf{u}_i} \\ \mathbf{u}_c \end{bmatrix} = \begin{bmatrix} \mathbf{f}_i \\ \boxed{\mathbf{f}_c} \end{bmatrix} \tag{4.36}$$

Unpacking the equations:

$$\begin{aligned}
\mathbf{K}_{ii}\boxed{\mathbf{u}_i} + \mathbf{K}_{ic}\mathbf{u}_c &= \mathbf{f}_i \\
\mathbf{K}_{ic}^T\boxed{\mathbf{u}_i} + \mathbf{K}_{cc}\mathbf{u}_c &= \boxed{\mathbf{f}_c}
\end{aligned} \tag{4.37}$$

Then first solve for the **unknown displacements**:

$$\begin{aligned}
\mathbf{K}_{ii}\boxed{\mathbf{u}_i} &= \mathbf{f}_u - \mathbf{K}_{ic}\mathbf{u}_c \\
\boxed{\mathbf{u}_i} &= \mathbf{K}_{ii}^{-1}\left(\mathbf{f}_i - \mathbf{K}_{ic}\mathbf{u}_c\right)
\end{aligned} \tag{4.38}$$

Lastly solve for the **unknown reaction forces**:

$$\boxed{\mathbf{f}_c} = \mathbf{K}_{ii}\mathbf{u}_i + \mathbf{K}_{ic}\mathbf{u}_c \tag{4.39}$$

Recovering Full Displacement Vector

The full displacement vector is then recovered:

$$\begin{aligned}
\mathbf{T} &= \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \overline{\mathbf{T}} \end{bmatrix} \\
\hat{\mathbf{u}} &= \begin{bmatrix} \mathbf{u}_i \\ \mathbf{u}_c \end{bmatrix} = \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix} \\
\mathbf{u}_g &= \mathbf{T}\hat{\mathbf{u}} \\
\begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \\ \mathbf{u}_s \end{bmatrix} &= \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \overline{\mathbf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{u}_u \\ \mathbf{u}_m \end{bmatrix}
\end{aligned} \tag{4.40}$$

4.2   RBE3

The first step is to calculate the **weighted COG** of the element, where

$$\mathbf{x}_{COG} = \frac{\sum w_i \mathbf{x}_i}{\sum w_i} \tag{4.41}$$

where w_i is a weighting factor of DOF i and $\mathbf{x}_i$ is the position of the node related with DOF i

Once the **COG** is found, the loading is transloated from the reference point $\mathbf{x}_{app}$ where it is applied to the **COG** by performing a equilibrium equivalency.

$$\mathbf{F}_{COG} = \mathbf{F}_{app} \tag{4.42}$$

$$\mathbf{M}_{COG} = \mathbf{M}_{app} + \mathbf{r} \times \mathbf{F}_{app} \tag{4.43}$$

Then the $\mathbf{F}_{COG}$ is reacted (shared) between all master nodes relative to their weight.

$$\mathbf{F}_i = \mathbf{F}_{COG} \frac{w_i}{\sum w_i} \tag{4.44}$$

The moment $\mathbf{M}_{COG}$ is assumed to be reacted (shared) by the mater DOFs according to the relative weight of each one of them and the relative position with regards to the COG. The reaction force that compensates the moment is in the direction perpendicular to the vector joining the COG with the position of the node where the master DOFs are:

$$\mathbf{P}_i = -w_i \frac{\mathbf{M}_{COG} \times \mathbf{r}_i}{\sum w_i.\mathbf{r}_i.\mathbf{r}_i} \tag{4.45}$$

where $\mathbf{r}_i$ is the vector from the COG position to the i-th node $\mathbf{r}_i = \mathbf{x}_i - \mathbf{x}_{COG}$.

The total reaction force on the master DOFs is the sum $\mathbf{F}_i + \mathbf{P}_i$ for each master DOF.

This force is not dependent on the stiffness of the model. It is only dependent on the weighting factors and the relative postion of the master nodes.

To get the **slave** node displacements:

(a) **COG** position

(b) $\mathbf{r}_i$ vectors

(c) compute the weighted average of the master displacements

(d) an unknown rotation is assumed and the displacements on the master nodes are written as if they were calculated with a rigid body movement with the weighted average displacements and the unknown rotations for each node i:

$$\mathbf{U}_s = \frac{\sum w_i \mathbf{U}_i}{\sum w_i} + \mathbf{R}_s \times \mathbf{r}_i \qquad (4.46)$$

where $\mathbf{U}$ are translational displacements and $\mathbf{R}$ are rotational displacements.

Chapter 5

Governing Equations

5.1 Material Matrices

Problem	Displacement components	Strain vector $\boldsymbol{\epsilon}^T$	Stress vector $\boldsymbol{\tau}^T$
Bar	u	$[\epsilon_{xx}]$	$[\sigma_{xx}]$
Beam	w	$[\kappa_{xx}]$	$[m_{xx}]$
Plane stress	u, v	$[\epsilon_{xx}, \epsilon_{yy}, \gamma_{xy}]$	$[\sigma_{xx}, \sigma_{yy}, \tau_{xy}]$
Plane strain	u, v	$[\epsilon_{xx}, \epsilon_{yy}, \gamma_{xy}]$	$[\sigma_{xx}, \sigma_{yy}, \tau_{xy}]$
Axisymmetric	u, v	$[\epsilon_{xx}, \epsilon_{yy}, \gamma_{xy}, \epsilon_{zz}]$	$[\sigma_{xx}, \sigma_{yy}, \tau_{xy}, \sigma_{zz}]$
3D	u, v, w	$[\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz}, \gamma_{xy}, \gamma_{yz}, \gamma_{zx}]$	$[\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \tau_{xy}, \tau_{yz}, \tau_{zx}]$
Plate bending	w	$[\kappa_{xx}, \kappa_{yy}, \kappa_{xy}]$	$[m_{xx}, m_{yy}, m_{xy}]$

Notation: $\epsilon_{xx} = \frac{\partial u}{\partial x}$, $\epsilon_{yy} = \frac{\partial v}{\partial y}$, $\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}$, ..., $\kappa_{xx} = \frac{\partial^2 w}{\partial x^2}$, $\kappa_{yy} = \frac{\partial^2 w}{\partial y^2}$, $\kappa_{xy} = \frac{\partial^2 w}{\partial x \partial y}$.

Table 5.1: Corresponding kinematic and static variables in various problems.

Problem	Material matrix C
Bar	E
Beam	EI
Plane stress	$\dfrac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$
Plane strain	$\dfrac{E(1-\nu)}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix}$
Axisymmetric	$\dfrac{E(1-\nu)}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & 1 & 0 & \frac{\nu}{1-\nu} \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 1 \end{bmatrix}$
3D	$\dfrac{E(1-\nu)}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix}$
Plate bending	$\dfrac{Eh^3}{12(1-\nu^2)}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$

Notation: E = Young's modulus, ν = Poisson's ratio, h = thickness of plate, I = moment of inertia

Table 5.2: Generalised stress-strain matrices for isotropic materials

Chapter 6

Dynamics

6.1 Function properties

Sine and Cosine functions

Let a fucntion $f(t)$ be in the form:

$$f(t) = A_1 \cos \omega t + B_i \sin \omega t \tag{6.1}$$

then the following relation applies:

$$f(t) = A \cos (\omega t - \phi)$$
$$A = \sqrt{A_1^2 + B_1^2}$$
$$\phi = \tan^{-1} \left(\frac{B_i}{A_i} \right) \tag{6.2}$$

also:

$$\cos (\omega t - \phi) = \cos \omega t \cos \phi + \sin \omega t \sin \phi$$
$$\sin (\omega t - \phi) = \sin \omega t \cos \phi + \cos \omega t \sin \phi \tag{6.3}$$

6.2   SDOF system

6.2.1   Definition

Let:

$$m\ddot{u}(t) + c\dot{u}(t) + ku(t) = f(t) \tag{6.4}$$

or:

$$ma(t) + cv(t) + ku(t) = f(t) \tag{6.5}$$

be the **SDOF equation of motion**, then the following applies:

$$
\begin{aligned}
\omega^2 &= \frac{k}{m} \\
\omega &= \sqrt{\frac{k}{m}} \\
f &= \frac{\omega}{2\pi} \\
T &= \frac{1}{f}
\end{aligned}
\tag{6.6}
$$

where:
ω is radial eigenfrequency [rad/s],
f is eigenfrequency [Hz] and
T is period [s]

The simplest vibratory syslem can be described by a single mass connected to a spring (and possibly a dashpot). The mass is allowed to travel only along the spring elongation direction. Such systems are called *Single Degree-of-Freedom* (**SDOF**) systems and are shown in the figure.

6.2.2   Equation of Motion for SDOF Systems

SDOF vibration can be analyzed by **Newton's second law of motion**, $F = ma$. The analysis can be easily visualised with the aid of a **free body diagram**,
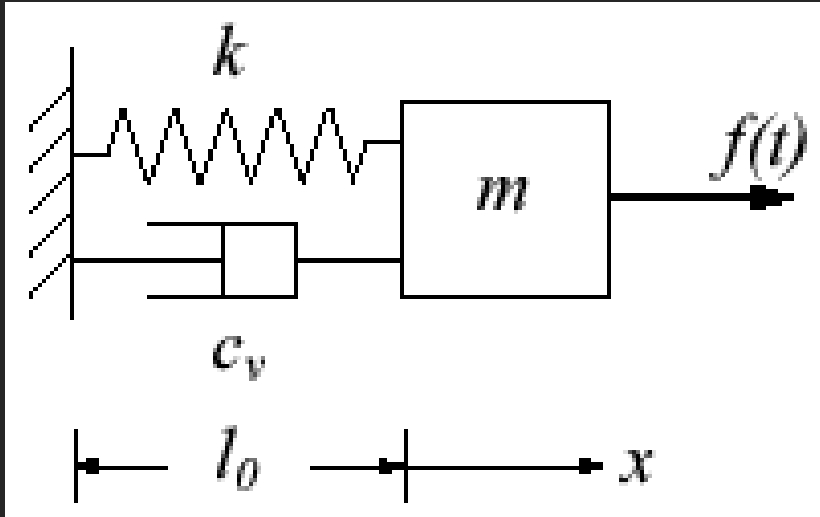
Figure 6.1: SDOF system with a linear spring and dashpot

The resulting equation of motion is a **second order, non-homegeneous, ordinary differential equation**:

$$m\ddot{u} + c\dot{u} + ku = f(t) \begin{cases} u(t=0) = u_0 \\ \dot{u}(t=0) = \dot{u}_0 \end{cases} \tag{6.7}$$

with the initial conditions u_0 and $\dot{u}_0$.

6.2.3 Time Solution for Unforced Undamped SDOF Systems

https://www.efunda.com/formulae/vibrations/sdof_free_undamped.cfm

The equation of motion derived in the definition can be simplified to:

$$m\ddot{u} + ku = 0 \begin{cases} u(t=0) = u_0 \\ \dot{u}(t=0) = \dot{u}_0 \end{cases} \tag{6.8}$$
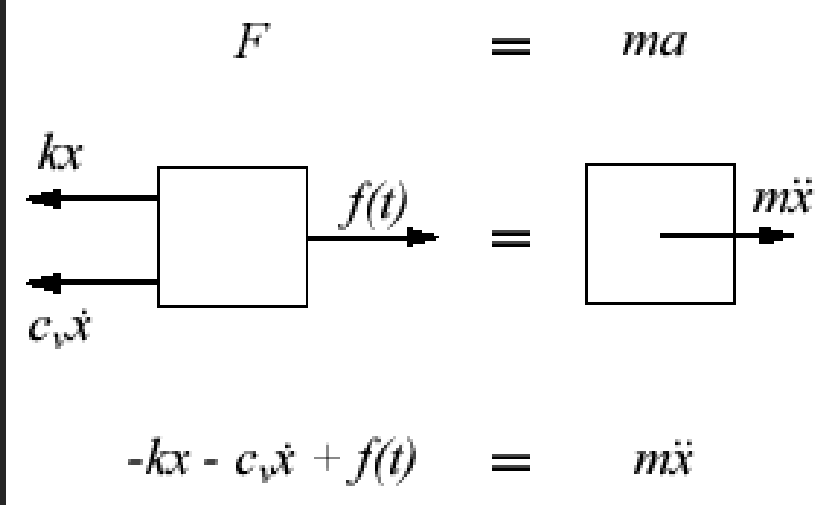
Figure 6.2: SDOF Free Body Diagram

This equation of motion is a **second order, homegeneous, ordinary differential equation** (ODE). If the mass and spring stiffness are constants, the ODE becomes a **linear, homogeneous ODE with constant coefficients** and can be solved by the Characteristic Equation method. The characteristic equation for this problem is:

$$ms^2 + k = 0 \qquad (6.9)$$

which determines the 2 independent roots for the undamped vibration problem. The final solution (that contains the 2 independent roots from the characteristic equation and satisfies the initial conditions) is:

$$
\begin{aligned}
u(t) &= c_1 e^{i\omega_n t} + c_2 e^{-\omega_n t} \\
&= d_1 \cos \omega_n t + d_2 \sin \omega_n t \\
\implies u(t) &= u_0 \cos \omega_n t + \frac{\dot{u}_0}{\omega_n} \sin \omega_n t
\end{aligned}
\qquad (6.10)
$$

The natural frequency ω_n is defined by:

$$\omega_n = \sqrt{\frac{k}{m}} \qquad (6.11)$$

and depends only on the system mass and the spring stiffness (i.e. any damping will not change the natural frequency of a system).

Alternatively, the solution may be expressed by the equivalent form,

$$u(t) = A_0 \cos\left(\omega_n t - \phi_0\right) \qquad (6.12)$$

where the amplitude A_0 and the intial phase ϕ_0 are given by:

$$A_0 = \sqrt{u_0^2 + \left(\frac{\dot{u}_0}{\omega_n}\right)^2}$$

$$\phi_0 = \tan^{-1}\frac{\dot{u}_0}{u_0 \omega_n} \qquad (6.13)$$
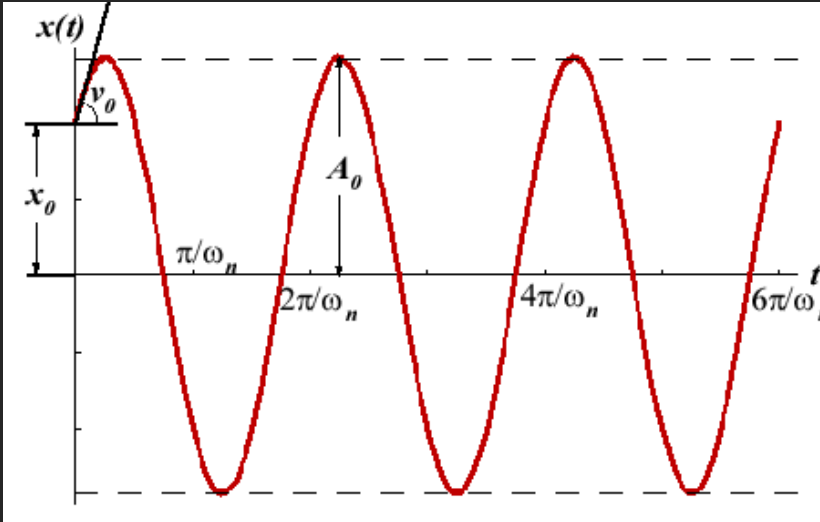
Figure 6.3: SDOF Undamped sample time behavior

Note that an assumption of zero damping is typically not accurate. In reality, there almost always exists some resistance in vibratory systems. This reistance will damp the vibration and dissipate energy, the oscillatory motion caused by the initial disturbance will eventually be reduced to zero.

6.2.4   Time Solution for Unforced Damped SDOF Systems

https://www.efunda.com/formulae/vibrations/sdof_free_damped.cfm
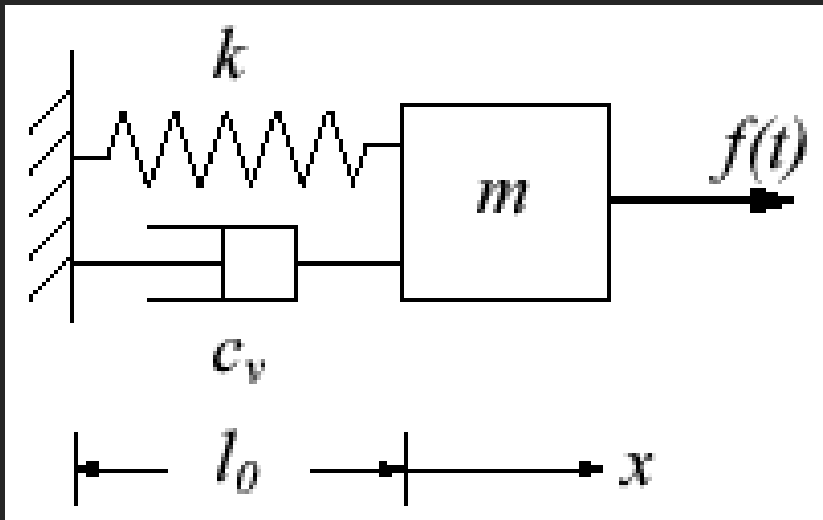
Figure 6.4: SDOF system with a linear spring and dashpot

Damping that produces a damping force proportional to the mass's velocity is commonly referred to as *viscous damping*, and is denoted graphically by a dashpot.

For an unforced damped **SDOF** system, the general equation of motion becomes:

$$m\ddot{u} + c\dot{u} + ku = 0 \begin{cases} u(t = 0) = u_0 \\ \dot{u}(t = 0) = \dot{u}_0 \end{cases} \qquad (6.14)$$

Tis equation of motion is a **second order, homogeneous, ODE**. If all parameters (mass, stiffness and viscous damping) are constants, the **ODE** becomes a **linear ODE with constant coefficients** and can be solved by the *Characteristic Equation method*. The characteristic equation for this problem is:

$$ms^2 + s_v s + k = 0 \tag{6.15}$$

which determines the 2 independent roots for the damped vibration problem. The roots to the characteristic equation fall into one of the following 3 cases:

- If $c_v^2 - 4mk < 0$, the system is termed **underdamped**. The roots of the characteristic equation are complex conjugates, corresponding to *oscillatory motion* with an *exponential decay* in amplitude.

- If $c_v^2 - 4mk = 0$, the system is termed **critically-derdamped**. The roots of the characteristic equation are repeated, corresponding to *simple decay motion* with at most *one overshoot* of the systems resting position.

- If $c_v^2 - 4mk > 0$, the system is termed **overderdamped**. The roots of the characteristic equation are prurely real and distinct, corresponding to *simple exponentially decaying motion*.

To simplify the solutions coming up, we define the critical damping c_c, the damping ratio ζ, and the damped vibration frequency ω_d as:

$$
\begin{aligned}
c_c &= 2m\sqrt{\frac{k}{m}} = 2m\omega_n \\
\zeta &= \frac{c_v}{c_c} \\
\omega_d &= \sqrt{1 - \zeta^2}\,\omega_n
\end{aligned}
\tag{6.16}
$$

where the natural frequency of the system ω_n is given by:

$$\omega_n = \sqrt{\frac{k}{m}} \tag{6.17}$$

Note that ω_d will equal ω_n when the damping of the system is zero (i.e. undamped). The time solutions for the free **SDOF** system is presented below for each of the three case scenarios.

Time Solution of Unforced Underdamped SDOF Systems

When $c_v^2 - 4mk < 0$ (equivalent to $\zeta < 1$ or $c_v < c_c$), the characteristic equation has a pair of complex conjugate roots. The displacement solution for this kind of system is:

$$
\begin{aligned}
u(t) &= c_1 e^{-\zeta + i\sqrt{1-\zeta^2}\omega_n t} + c_2 e^{-\zeta - i\sqrt{1-\zeta^2}\omega_n t} \\
&= e^{-\zeta\omega_n t} \left[d_1 \cos\left(\omega_d t\right) + d_2 \sin\left(\omega_d t\right) \right] \\
\implies u(t) &= \underbrace{e^{-\zeta\omega_n t}}_{exponential\ decay} \underbrace{\left[u_0 \cos\left(\omega_d t\right) + \frac{\dot{u}_0 + \zeta\omega_n u_0}{\omega_d} \sin\left(\omega_d t\right) \right]}_{periodic\ motion}
\end{aligned} \tag{6.18}
$$

An alternate but equivalent solution is given by:

$$
u(t) = A_0 \underbrace{e^{-\zeta\omega_n t}}_{exponential\ decay} \underbrace{\cos\left(\omega_d t - \phi_0\right)}_{periodic\ motion} \tag{6.19}
$$

Note that the displacement amplitude decays exponentially (i.e. the natural logarithm of the amplitude ratio for any two displacements separated in time by a constant ratio is a constant):

$$
\begin{aligned}
\frac{A_k}{A_{k+1}} &= \frac{A_0 e^{-\zeta\omega_n (kT_d)} \cos\left(\phi_0\right)}{A_0 e^{-\zeta\omega_n [(k+1)T_d]} \cos\left(\phi_0\right)} \\
&= \frac{e^{-\zeta\omega_n (kT_d)}}{e^{-\zeta\omega_n [(k+1)T_d]}} \\
&= e^{\zeta\omega_n T_d}
\end{aligned} \tag{6.20}
$$

$$
\implies \ln\left(\frac{A_k}{A_{k+1}}\right) = \zeta\omega_n T_d = \zeta\omega_n \frac{2\pi}{\omega_d} = \frac{2\pi\zeta}{\sqrt{1-\zeta^2}}
$$

where $T_d = \frac{1}{f_d} = \frac{2\pi}{\omega_d}$ is the period of the damped vibration.
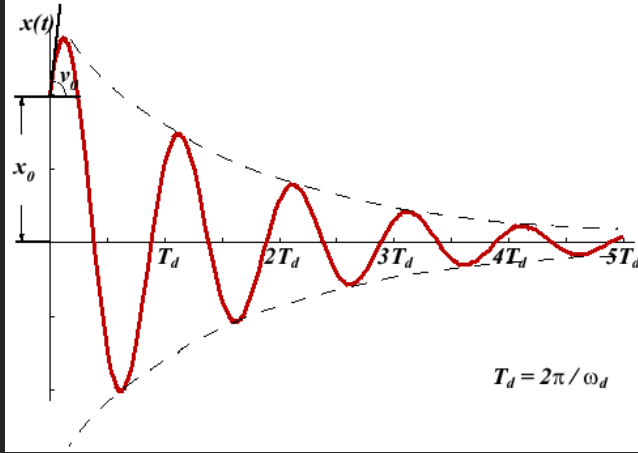
Figure 6.5: The displacement plot of an underdamped system

Time Solution of Unforced Critically-Damped Systems

When $c_v^2 - 4mk = 0$ (equivalent to $\zeta = 1$ or $c_v = c_c$), the characteristic equation has repeated real roots. The displacement solution for this kind of system is:

$$\begin{aligned} u(t) &= (c_1 + c_2 t)\, e^{-\omega_n t} \\ \implies u(t) &= e^{-\omega_n t} \left[u_0 + (v_0 + \omega_n u_0)\, t \right] \end{aligned} \tag{6.21}$$

The critical damping factor c_c can be interpreted as the **minimum damping** that results in non-periodic motion (i.e. simple decay).

The displacement decays to a negligible level after one natural period T_n. Note that if the initial velocity v_0 is negative while the initial displacement u_0 is positive, there will exist one overshoot of the resting position in the displacement plot.

Time Solution of Unforced Overdamped SDOF Systems

When $c_v^2 - 4mk > 0$ (equivalent to $\zeta > 1$ or $c_v > c_c$), the characteristic equation has two distinct real roots. The displacement solution for this
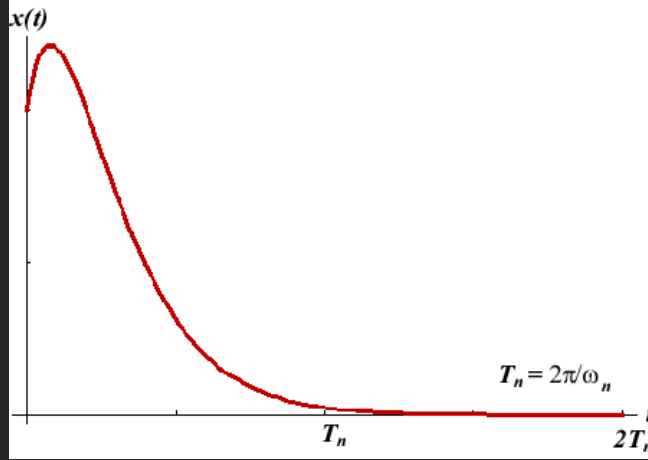
Figure 6.6: The displacement plot of a critically-damped system with positive initial displacement and velocity.

kind of system is:

$$u(t) = c_1 e^{\left(-\zeta + \sqrt{\zeta^2 - 1}\right)\omega_n t} + c_2 e^{\left(-\zeta - \sqrt{\zeta^2 - 1}\right)\omega_n t}$$

$$\implies u(t) = \frac{u_0 \omega_n \left(\zeta + \sqrt{\zeta^2 - 1}\right) + v_0}{2\omega_n \sqrt{\zeta^2 - 1}} e^{\left(-\zeta + \sqrt{\zeta^2 - 1}\right)\omega_n t} +$$

$$+ \frac{-u_0 \omega_n \left(\zeta - \sqrt{\zeta^2 - 1}\right) - v_0}{2\omega_n \sqrt{\zeta^2 - 1}} e^{\left(-\zeta - \sqrt{\zeta^2 - 1}\right)\omega_n t} \qquad (6.22)$$

The motion of an overdamped system is non-periodic, regardless of the inital conditions. The larger the damping, the longer time to decay from an initial disturbance.

If the system is heavily damped, $\zeta \gg 1$, the displacement solution takes the approximate form:

$$u(t) \approx u_0 + \frac{v_0}{2\zeta\omega_n}\left(1 - e^{-2\zeta\omega_n t}\right) \qquad (6.23)$$
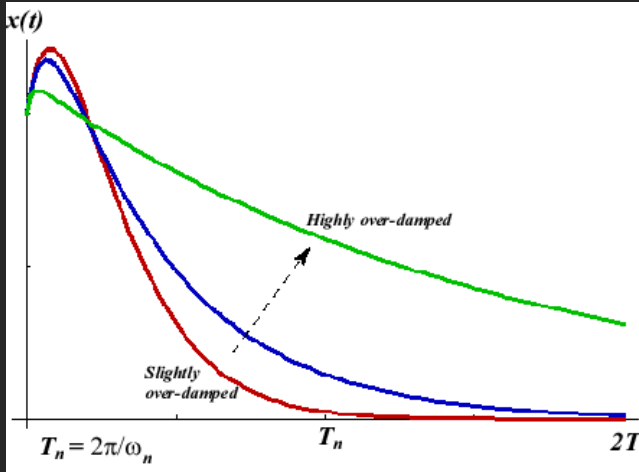
Figure 6.7: The displaceent plot of an overdamped system response

6.2.5 SDOF Systems under Harmonic Excitaion

https://www.efunda.com/formulae/vibrations/sdof_harmo.cfm

When a **SDOF** System is forced by $f(t)$, the solution for the displacement $x(t)$ consists of two parts: the *complimentary solution*, and the *particular solution*. The complimentary solution for the problem is given by the **free vibration of an unforced SDOF System**. The **particular solution** depends on the nature fo the forcing function.

When the forcing function is **harmonic** (i.e. it consits of at most a **sine** and **cosine** at the same **frequency**, a quantity that can be expressed by the complex exponentional $e^{i\omega t}$), the method if **Undetermined Coefficients** can be used to find the particular solution. Non-harmonic forcing functions are handled by other techniques.

Consider the SDOF system forced by the harmonic function $f(t)$.

The particular solution for this problem is found to be:

$$u_p(t) = \frac{f_0}{(k - m\omega^2) + ic\omega} e^{i\omega t} \qquad (6.24)$$
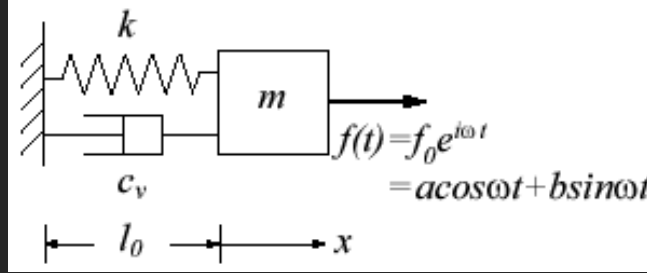
Figure 6.8: SDOF System forced by the harmonic function $f(t)$

The general solution is given by the sum of the complimentary and particular solutions multiplied by two weighting constants c_1 and c_2,

$$u(t) = c_1 u_c(t) + c_2 u_p(t) \tag{6.25}$$

The values of c_1 and c_2 are found by matching $u(t = 0)$ to the initial conditions.

Undamped SDOF Systems under Harmonic Excitation

For an undamped system ($c_v = 0$) the total displacement solution is:

$$
\begin{aligned}
u(t) &= d_1 \cos \omega_n t + d_2 \sin \omega_n t + \frac{f_0}{k - m\omega^2} e^{i\omega t} \\
\implies u(t) &= \left(u_0 - \frac{f_0}{k - m\omega^2} \right) \cos \omega_n t \\
&+ \left(\frac{v_0 - \frac{i\omega f_0}{k - m\omega^2}}{\omega_n} \right) \sin \omega_n t \\
&+ \frac{f_0}{k - m\omega^2} e^{i\omega t}
\end{aligned}
\tag{6.26}
$$

If the forcing frequency is close to the natural frequency, $\omega \approx \omega_n$, the system will exhibit **resonance** (very large displacements) due to near-zeros in the denominators of $u(t)$.

When the forcing frequency is equal to the natural frequency, we cannot use the $u(t)$ given above as it would give divide-by-zero. Instead we must use **L'Hôspital's Rule** to derive a solution free of zeros in the denominators.

$$u(t) = u_0 \cos \omega_n t + \frac{v_0}{\omega_n} \sin \omega_n t$$

$$+ \lim_{\omega \to \omega_n} \left\{ \frac{f_0}{k - m\omega^2} \left(e^{i\omega t} - \cos \omega_n t - i\omega \sin \omega_n t \right) \right\} \tag{6.27}$$

$$= u_0 \cos \omega_n t + \frac{v_0}{\omega_n} \sin \omega_n t - \frac{f_0 \omega_n}{2k} \left(it e^{i\omega)nt} - i \sin \omega_n t \right)$$

To simplify $u(t)$, let's assume that the driving force consists only of the cosine function $f(t) = f_0 \cos \omega t$.
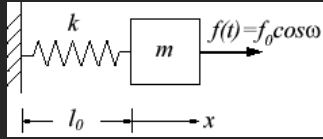
Figure 6.9: SDOF System forced by the harmonic function $f(t) = f_0 \cos \omega t$

The displacement solution reduces to:

$$u(t) = u_0 \cos \omega_n t + \frac{v_0}{\omega_n} \sin \omega_n t$$

$$+ \lim_{\omega \to \omega_n} \left\{ \frac{f_0}{k - m\omega^2} \left(\cos \omega t - \cos \omega_n t \right) \right\}$$

$$= \underbrace{u_0 \cos \omega_n t + \frac{v_0}{\omega_n} \sin \omega_n t}_{free\ vibration\ (complementary)} + \underbrace{\frac{f_0 \omega_n t}{2k}}_{\substack{amplitude \\ linearly \\ increased}} \sin \omega_n t \tag{6.28}$$

This solution contains one term multiplied by t. This term will cause the displacement amplitude to increase linearly wit time as the forcing function pumps energy into the system.
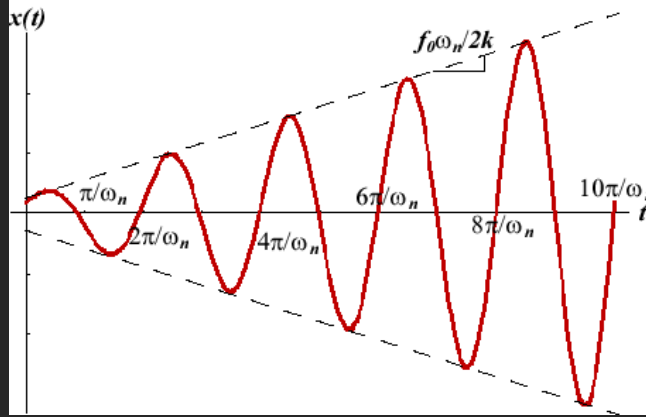
Figure 6.10: Resonance response of undamped SDOF with harmonic excitation

The maximum displacement of an undamped system forced at its reso-
nant frequency will increase unbounded according to the solution for $u(t)$
above. However, real systems will inject additional physics once displace-
ments become large enough. These additional physics (nonlinear plastic
deformation, heat transfer, buckling etc.) will serve to limit the maxi-
mum displacement exhibited by the system, and allow one to escape the
"sudden death" impression that such systems will immediately fail.

Damped SDOF Systems under Harmonic Excitation

$$m\ddot{u} + c\dot{u} + ku = F_0 \cos(\omega t) \tag{6.29}$$

then **steady state** solution is in the form:

$$u_{s.s.} = u_0 \cos(\omega t - \phi) \tag{6.30}$$

If we plug the **steady state** solution to the **equation of motion**, we get:

$$u_o \left[\left(k - m\omega^2 \right) \cos\omega t - \phi - c\omega \sin(\omega t - \phi) \right] = F_0 \cos(\omega t) \quad | \times \frac{1}{k} \tag{6.31}$$
$$u_o \left[\left(1 - \frac{\omega^2}{\omega_n^2} \right) \cos\omega t - \phi - 2\zeta \frac{\omega}{\omega_n} \sin(\omega t - \phi) \right] = \frac{F_0}{k} \cos(\omega t)$$

because:

$$\frac{k}{m} = \omega_n \tag{6.32}$$

$$\zeta = \frac{c}{c_c} \tag{6.33}$$

$$c = 2m\sqrt{\frac{k}{m}} = 2m\omega_n \tag{6.34}$$

$$\frac{c}{k} = \frac{\zeta c_c}{k} = \frac{\zeta 2m\omega_n}{k} = \frac{2\zeta\omega_n^2}{omega_n^2} = \frac{2\zeta}{\omega_n} \tag{6.35}$$

after utilising the sine and cosine relations:

$$\cos{(\omega t - \phi)} = \cos\omega t \cos\phi + \sin\omega t \sin\phi$$
$$\sin{(\omega t - \phi)} = \sin\omega t \cos\phi + \cos\omega t \sin\phi \tag{6.36}$$

we get a set of two equatons:

$$u_o \left[\left(1 - \frac{\omega^2}{\omega_n^2} \right) \cos\phi + 2\zeta\frac{\omega}{\omega_n} \sin\phi \right] \cos\omega t = \frac{F_0}{k} \cos{(\omega t)}$$
$$u_o \left[\left(1 - \frac{\omega^2}{\omega_n^2} \right) \sin\phi - 2\zeta\frac{\omega}{\omega_n} \cos\phi \right] \sin\omega t = 0 \tag{6.37}$$

and when solving just for amplitudes this reduces to a set of two algebraic equations which is then solved for u_0 and ϕ:

$$u_o \left[\left(1 - \frac{\omega^2}{\omega_n^2} \right) \cos\phi + 2\zeta\frac{\omega}{\omega_n} \sin\phi \right] = \frac{F_0}{k}$$
$$u_o \left[\left(1 - \frac{\omega^2}{\omega_n^2} \right) \sin\phi - 2\zeta\frac{\omega}{\omega_n} \cos\phi \right] = 0 \tag{6.38}$$

The solution (**transfer function**) is finally:

$$u_0 = \frac{\frac{F_0}{k}}{\left[\left(1 - \frac{\omega^2}{\omega_n^2} \right)^2 + \left(2\zeta\frac{\omega}{\omega_n} \right)^2 \right]^{\frac{1}{2}}}$$
$$\phi = \tan^{-1}\left(\frac{2\zeta\frac{\omega}{\omega_n}}{1 - \frac{\omega^2}{\omega_n^2}} \right) \tag{6.39}$$

where:

u_0 is displacement amplitude

$\frac{F_0}{k}$ is displacement from static simulation.

$$H_0 = \frac{\frac{F_0}{k}}{\left[\left(1 - \frac{\omega^2}{\omega_n^2}\right)^2 + \left(2\zeta\frac{\omega}{\omega_n}\right)^2\right]^{\frac{1}{2}}} \tag{6.40}$$

6.3   Damping

6.3.1   Rayleigh Damping

https://www.simscale.com/knowledge-base/rayleigh-damping-coefficients/

Let **Rayleigh Damping** be defined as:

$$c = \alpha k + \beta m \qquad (6.41)$$

where:
c is damping value [-],
α is stiffness dependent damping coefficient,
β is inertia (mass) dependent damping coefficient,
m is mass and
k is stiffness.

Thus, substituting this relation to the equation of motion:

$$
\begin{aligned}
m\ddot{(u)} + (\alpha k + \beta m)\,\dot{u} + ku &= f(t) & &| \times \frac{1}{m} \\
\ddot{u} + \left(\alpha\frac{k}{m} + \beta\frac{m}{m}\right)\dot{u} + \frac{k}{m}u &= \frac{f(t)}{m} & &|\frac{k}{m} = \omega_n^2 \\
\ddot{u} + \left(\alpha\omega_n^2 + \beta\right)\dot{u} + \omega_n^2 u &= \frac{f(t)}{m}
\end{aligned}
\qquad (6.42)
$$

Damping Ratio:

$$
\begin{aligned}
\zeta &= \frac{c}{2m\omega_n} \\
\zeta &= \frac{1}{2m\omega_n}\left(\alpha k + \beta m\right) \\
\zeta &= \frac{1}{2}\left(\alpha\omega_n + \frac{\beta}{\omega_n}\right)
\end{aligned}
\qquad (6.43)
$$

where:
ζ is damping ratio,
c is damping value [-],

m is mass and

ω_n is natural frequency [rad/s]

Substituting back:

$$\ddot{u} + 2\zeta\omega_n\dot{u} + \omega_n^2 ku = \frac{f(t)}{m} \tag{6.44}$$

When **Damping is proportional to inertia**:

In this case, the stiffness coefficient $\alpha = 0$ and thus:

$$\zeta = \frac{\beta}{2\omega_n} \tag{6.45}$$

For a given constant value of β, it is seen that the damping is inversely proportional to the natural frequency, as shown in the illustration:
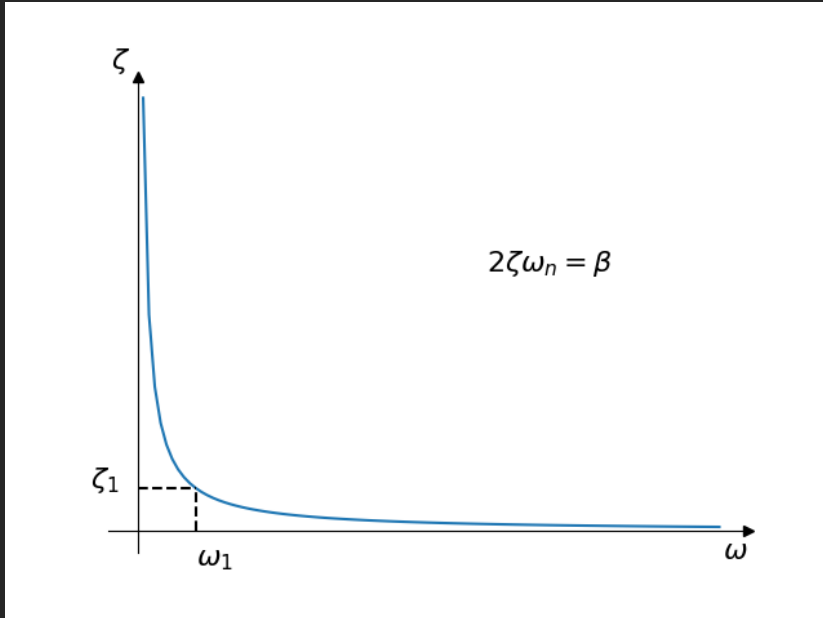
Figure 6.11: Schematic of damping proportional to inertia

Moreover, if one computes β from the damping ration ζ_1 at a given natural frequency ω_1, all the natural frequencies below it will be amplified and the frequencies above it will be attenuated. The effect is more dramatic the farther the frequencies are from the reference value.

When **Damping is proportional to stiffness**:

In this case, the mass coefficient $\beta = 0$ and thus:

$$\zeta = \frac{1}{2}\alpha\omega_n \tag{6.46}$$

It is seen that, contrary to the first case, here the damping id directly proportional to the natural frequency:
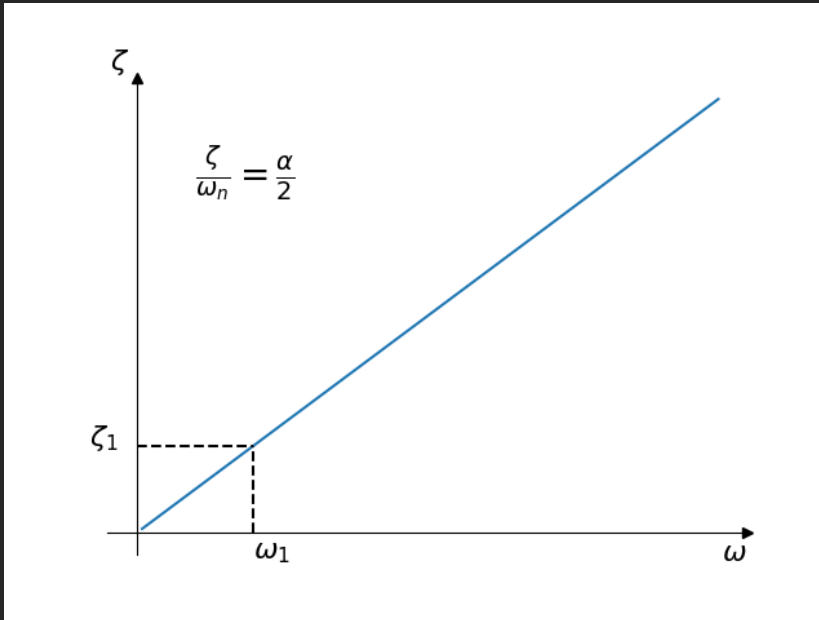
Figure 6.12: Schematic of damping proportional to stiffness

If one computes α from the damping ratio ζ_1 at a natural frequency ω_1, then the natural frequencies below will be attenuated and the frequencies above will be amplified.

General Case:

In the case of using the model with two parameters, the proportionality of damping against frequency is convex:
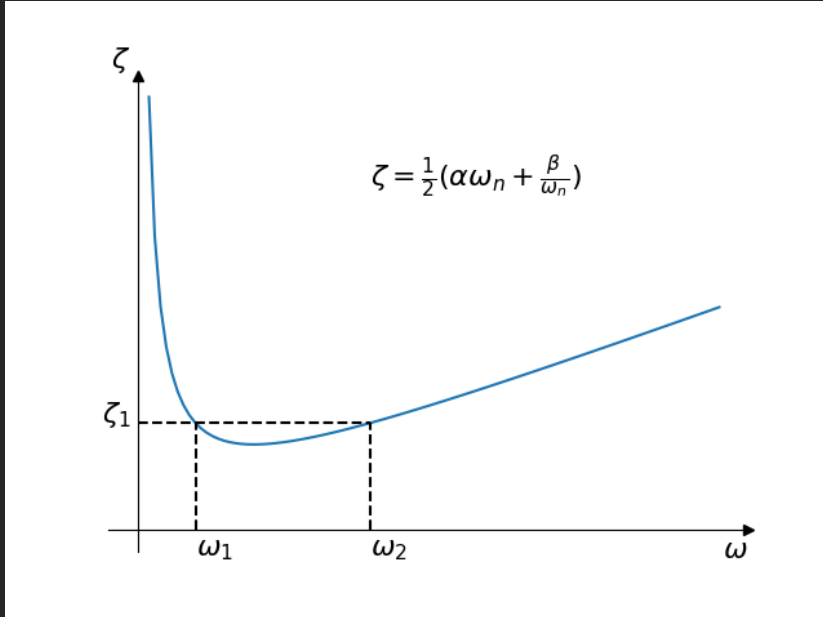
Figure 6.13: Schematic of full damping model

In this case one needs two damping ratios and two natural frequencies to create a pair of equations and solve for α and β. The model gives some flexibility on where to place the natural frequencies, but in general, frequencies too far away from the ones used in the computation will be amplified.

In the particular case of using equal damping ratios for the two frequencies, it is important to note that the damping ratio **will not be constant inside the range** defined by the sample points, but the inner frequencies will be attenuated. That is, the inner frequencies will have a lower damping ratio.

Computing the Rayleigh Damping Coefficients

In the most common case, a transient response curve frim the system is
obtained and the damping ratio ζ_1 is determined for the lowest natural
frequency ω_1 by measuring the (logarithmic) attenuation of successive
peaks.
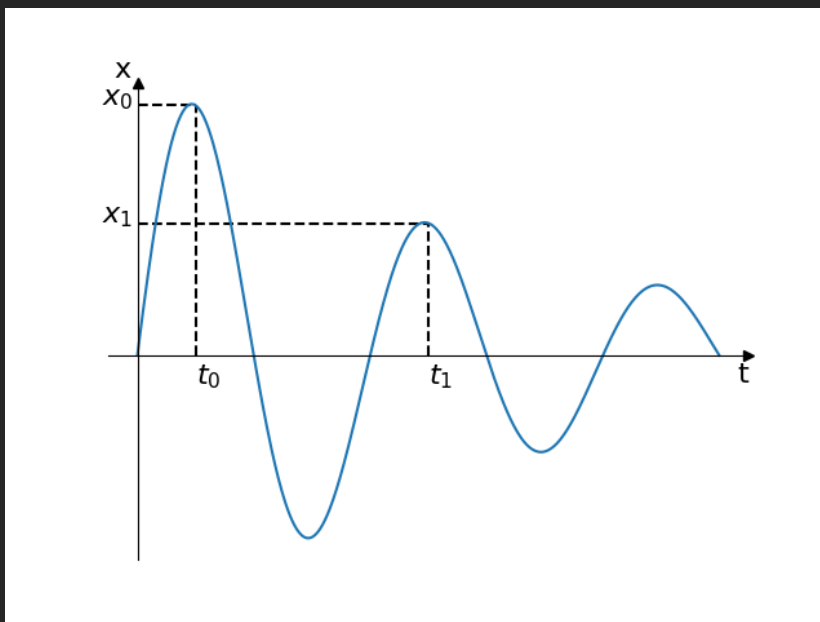
Figure 6.14: Determination of the damping ration from the logarithmic decay

$$\zeta = \frac{\delta}{\sqrt{\delta^2 + (2\pi)^2}}$$

$$\delta = \ln \frac{x_0}{x_1} \tag{6.47}$$

$$f = \frac{1}{T} = \frac{1}{t_1 - t_0}$$

$$\omega = 2\pi f$$

It is then most common to assume the case of damping proportional to

the stiffness, that is, $\beta = 0$, and the α stiffness coefficient is computed from:

$$\alpha = \frac{2\zeta_1}{\omega_1} = \frac{\zeta + 1}{\pi f_1} \tag{6.48}$$

If the knowledge on the system indicates the case of damping decreasing with the frequency, then one can assume the case of damping proportional to the inertia, where $\alpha = 0$ and determine the mass coefficient β:

$$\beta = 2\zeta_1\omega_1 = 4\pi\zeta_1 f_1 \tag{6.49}$$

If there is not such test data or knowledge of the system, or if one wishes to apply an approximate damping ration over a range of frequencies, then we can use the general case and build a system of two equations:

$$\begin{aligned} \zeta_1 &= \frac{1}{2}\left(\alpha\omega_1 + \frac{\beta}{\omega_1}\right) \\ \zeta_2 &= \frac{1}{2}\left(\alpha\omega_2 + \frac{\beta}{\omega_2}\right) \end{aligned} \tag{6.50}$$

Then solve for the unknown coefficients, keeping in mint the considerations given above for the general case and the influence of the model on natural frequencies inside and outside the range of interest. That is, perhaps one wants to achieve a mean damping ratio over the range, then compensate the attenuation by modifying the input damping rations, or by performing some least-squares approximation from more than two frequency points.

6.4 Modal decomposition

6.4.1 Definition

Let:

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{C}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{F}\,(t) \tag{6.51}$$

be the **equation of motion**, then:
$\mathbf{\Lambda}$ is a **spectral** matrix
$\mathbf{\Psi}$ is a **mode shape** matrix,

such that:

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_1^2 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n^2 \end{bmatrix} \tag{6.52}$$

where λ_n^2 is the n-th eigenvalue of the problem $(\mathbf{K} - \lambda^2 \mathbf{M})\mathbf{X} = 0$

and:

$$\mathbf{\Psi} = \begin{bmatrix} \boldsymbol{\psi}_1 & \cdots & \boldsymbol{\psi}_n \end{bmatrix} \tag{6.53}$$

where $\boldsymbol{\psi}_n = \{\psi_{n,1}, \dots, \psi_{n,m}\}^T$ is the eigenvector of the n-th eigenvalue λ_n^2.

The **displacement u** can be expressed as a linear combination of eigenvectors such that:

$$\begin{aligned} \mathbf{u}(t) &= \mathbf{\Psi}\mathbf{q}(t) \\ &= \begin{bmatrix} \boldsymbol{\psi}_1 & \cdots & \boldsymbol{\psi}_n \end{bmatrix} \mathbf{q}(t) \\ &= \begin{bmatrix} \psi_{1,1} & \cdots & \psi_{n,1} \\ \vdots & \vdots & \vdots \\ \psi_{1,m} & \cdots & \psi_{n,m} \end{bmatrix} \begin{bmatrix} q_1(t) \\ \vdots \\ q_n(t) \end{bmatrix} \end{aligned} \tag{6.54}$$

where $\mathbf{q}(t)$ is a vector of modal coefficients.

Then:

$$\dot{\mathbf{u}}(t) = \mathbf{\Psi}\dot{\mathbf{q}}(t)$$
$$\ddot{\mathbf{u}}(t) = \mathbf{\Psi}\ddot{\mathbf{q}}(t)$$

$$(6.55)$$

First we substitute $\mathbf{u} = \mathbf{\Psi}\mathbf{q}$ to the **equation of motion**:

$$\mathbf{M}\mathbf{\Psi}\ddot{q} + \mathbf{C}\mathbf{\Psi}\dot{q} + \mathbf{K}\mathbf{\Psi}\mathbf{q} = \mathbf{F}(t) \tag{6.56}$$

By **premultiplying** with $\mathbf{\Psi}^T$ we get:

$$\mathbf{\Psi}^T\mathbf{M}\mathbf{\Psi}\ddot{\mathbf{q}} + \mathbf{\Psi}^T\mathbf{C}\mathbf{\Psi}\dot{\mathbf{q}} + \mathbf{\Psi}^T\mathbf{K}\mathbf{\Psi}\mathbf{q} = \mathbf{\Psi}\mathbf{F}(t) = \mathbf{Q}(t) \tag{6.57}$$

where $\mathbf{Q}(t) = \mathbf{\Psi}\mathbf{F}(t)$ is called a **modal load**.

6.4.2   Orthonormalised Modal Base

Then one can exploit the properties of **orthonormalised eigenvectors**, where:

$$\boldsymbol{\psi}_r^T\boldsymbol{\psi}_s = 0 \Leftrightarrow r \neq s \tag{6.58}$$

The **mass**, **damping** and **stiffness** matrices are therefore reduced to:

$$\mathbf{\Psi}^T\mathbf{M}\mathbf{\Psi} = \begin{bmatrix} m_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & m_n \end{bmatrix}$$

$$\mathbf{\Psi}^T\mathbf{C}\mathbf{\Psi} = \begin{bmatrix} c_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & c_n \end{bmatrix} \tag{6.59}$$

$$\mathbf{\Psi}^T\mathbf{K}\mathbf{\Psi} = \begin{bmatrix} k_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & k_n \end{bmatrix}$$

where:
m_i is the **modal mass** of the i-th shape

c_i is the **modal damping** of the i-th shape
k_i is the **modal stiffness** of the i-th shape

Note:
When are the eigenvectors **orthonormalised** to **mass**, the above equations reduce to:

$$\boldsymbol{\Psi}^T \mathbf{M} \boldsymbol{\Psi} = \begin{bmatrix} 1.0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1.0 \end{bmatrix} = \mathbf{I}$$

$$\boldsymbol{\Psi}^T \mathbf{C} \boldsymbol{\Psi} = \begin{bmatrix} c_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & c_n \end{bmatrix} \tag{6.60}$$

$$\boldsymbol{\Psi}^T \mathbf{K} \boldsymbol{\Psi} = \begin{bmatrix} \lambda_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \lambda_n^2 \end{bmatrix} = \boldsymbol{\Lambda}^2$$

Mass normalisation:

$$\boldsymbol{\psi}_{i,mass} = \frac{\boldsymbol{\psi}_i}{\boldsymbol{\psi}_i^T \mathbf{m}_i \boldsymbol{\psi}_i} \tag{6.61}$$

where:
$\mathbf{m}_i$ is the i-th row of the **mass** matrix $\mathbf{M}$.

6.4.3   MDOF to SDOF

The MDOF **equation of motion** can be then rewritten as a set of **SDOF** equations:

$$\boldsymbol{\Psi}^T \mathbf{M} \boldsymbol{\Psi} \ddot{\mathbf{q}}(t) + \boldsymbol{\Psi}^T \mathbf{C} \boldsymbol{\Psi} \dot{\mathbf{q}}(t) + \boldsymbol{\Psi}^T \mathbf{K} \boldsymbol{\Psi} \mathbf{q}(t) = \boldsymbol{\Psi}^T \mathbf{F}(t)$$
$$\mathbf{M}_{\boldsymbol{\Psi}} \ddot{\mathbf{q}}(t) + \mathbf{C}_{\boldsymbol{\Psi}} \dot{\mathbf{q}}(t) + \mathbf{K}_{\boldsymbol{\Psi}} \mathbf{q}(t) = \mathbf{Q}(t) \tag{6.62}$$

Written explicitly:

$$
\begin{bmatrix} m_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & m_n \end{bmatrix} \begin{bmatrix} \ddot{q}_1(t) \\ \vdots \\ \ddot{q}_n(t) \end{bmatrix} +
$$
$$
\begin{bmatrix} c_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & c_n \end{bmatrix} \begin{bmatrix} \dot{q}_1(t) \\ \vdots \\ \dot{q}_n(t) \end{bmatrix} +   \tag{6.63}
$$
$$
\begin{bmatrix} k_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & k_n \end{bmatrix} \begin{bmatrix} q_1(t) \\ \vdots \\ q_n(t) \end{bmatrix} = \begin{bmatrix} Q_1(t) \\ \vdots \\ Q_n(t) \end{bmatrix}
$$

It is important to note that all modal matrices (mass, damping and stiffness) are purely diagonal (with simple enough damping), so the **MDOF** problem reduces to a series of **SDOF** problems:

$$
\begin{aligned}
m_1\ddot{q}_1(t) + c_1\dot{q}_1(t) + k_1 q_1(t) &= Q_1(t) \\
m_2\ddot{q}_2(t) + c_2\dot{q}_2(t) + k_2 q_2(t) &= Q_2(t) \\
&\vdots \\
m_n\ddot{q}_n(t) + c_n\dot{q}_n(t) + k_n q_n(t) &= Q_n(t)
\end{aligned}
\tag{6.64}
$$

which are solved separately and the results are **linearly** combined:

$$
\begin{aligned}
\mathbf{u} &= \sum_{i=1}^{n} \boldsymbol{\psi}_i q_i \\
\dot{\mathbf{u}} &= \sum_{i=1}^{n} \boldsymbol{\psi}_i \dot{q}_i \\
\ddot{\mathbf{u}} &= \sum_{i=1}^{n} \boldsymbol{\psi}_i \ddot{q}_i
\end{aligned}
\tag{6.65}
$$

Note: Above equation can be also written as:

$$
\begin{aligned}
\mathbf{u} &= \sum_{i=1}^{n} \boldsymbol{\psi}_i q_{u,i} \\
\mathbf{v} &= \sum_{i=1}^{n} \boldsymbol{\psi}_i q_{v,i} \\
\mathbf{a} &= \sum_{i=1}^{n} \boldsymbol{\psi}_i q_{a,i}
\end{aligned}
\tag{6.66}
$$

where:
u is displacement
v is velocity
a is acceleration
and
$\mathbf{q}_u$ is modal displacement
$\mathbf{q}_v$ is modal velocity
$\mathbf{q}_a$ is modal acceleration

6.4.4   Initial Conditions

To get the initial modal conditions, one simply:

$$
\begin{aligned}
\mathbf{q}_0 &= \boldsymbol{\Psi}^{-1} \mathbf{u}_0 \\
\dot{\mathbf{q}}_0 &= \boldsymbol{\Psi}^{-1} \dot{\mathbf{u}}_0 \\
\ddot{\mathbf{q}}_0 &= \boldsymbol{\Psi}^{-1} \ddot{\mathbf{u}}_0
\end{aligned}
\tag{6.67}
$$

where:
$\boldsymbol{\Psi}^{-1}$ is an inverse of eigenshape matrix $\Leftrightarrow$ all eigenvalues and eigenshapes
are used.

Note: This is not usually so. Normally one would use **reduced modal base**, where only the first **n** eigenmodes are used. Then $\mathbf{\Psi}^{-1}$ is written as $\mathbf{\Psi}^+$ and called the **pseudoinverse of mode shape matrix** defined (at least in **numpy** - *numpy.linal.pinv()*) as:

The pseudo-inverse of a matrix $\mathbf{A}$, denoted $\mathbf{A}^+$, is defined as: "the matrix that 'solves' [the least-squares problem] $\mathbf{Ax} = \mathbf{b}$," i.e., if $\bar{\mathbf{x}}$ is said solution, then $\mathbf{A}^+$ is that matrix such that $\bar{\mathbf{x}} = \mathbf{A}^+\mathbf{b}$.

It can be shown that if $\mathbf{Q}_1\mathbf{\Sigma}\mathbf{Q}_2^T = \mathbf{A}$ is the singular value decomposition of $\mathbf{A}$, then $\mathbf{A}^+ = \mathbf{Q}_2\mathbf{\Sigma}^+\mathbf{Q}_1^T$, where $\mathbf{Q}_{1,2}$ are orthogonal matrices, $\mathbf{\Sigma}$ is a diagonal matrix consisting of $\mathbf{A}$'s so-called singular values, (followed, typically, by zeros), and then $\mathbf{\Sigma}^+$ is simply the diagonal matrix consisting of the reciprocals of $\mathbf{A}$'s singular values (again, followed by zeros).

Chapter 7

Kinematics

7.1 Rigid Body Motion

A rigid freebody relation between **slave** (dependent) and **master** (independent) DOFs is calculated following the equation:

$$\begin{aligned}
\mathbf{T}_s &= \mathbf{T}_m + \mathbf{R}_s \times \mathbf{x} \\
\mathbf{R}_s &= \mathbf{R}_m
\end{aligned} \tag{7.1}$$

where $\mathbf{T}$ is a translations vector $= \{\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3\}$, $\mathbf{R}$ is a rotations vector $= \{\mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3\}$, $\mathbf{x}$ is a vector from slave (base) to the master node (tip) $= \{\mathbf{x}_s - \mathbf{x}_m, \mathbf{y}_s - \mathbf{y}_m, \mathbf{z}_s - \mathbf{z}_m\}$ and $\times$ is a vector (cross) product.

Chapter 8

Iterative Methods

8.1 second order ODE's

Most of FEM problems are **second order Ordinary Differential Equations**, e.g:

$$m\frac{u''(t)}{d^2t} + c\frac{u'(t)}{dt} + ku(t) = 0$$
$$m\ddot{u} + c\dot{u} + ku = 0 \tag{8.1}$$
$$ma + cv + ku = 0$$

while the iterative methods are for **first order ODE's** only. A **substitution** technique is therefore used to transform the **second order ODE's** to a set of **first order ODE's** and these are then solved sequentialy.

First we begin with a **second order ODE** in the form of:

$$a\frac{y''(x)}{d^2x} + b\frac{y'(x)}{dx} + cy = d, \quad with \quad y(0) = y_0 \quad and \quad \frac{y'(0)}{dx} = y_1 \tag{8.2}$$

By stating:

$$\frac{dy}{dx} = z \tag{8.3}$$

and substituting back:

$$a\frac{dz}{dx} + bz + cy = d \tag{8.4}$$

now we can write the following set of equations:

$$
\begin{aligned}
z &= \frac{dy}{dx} \\
\frac{dz}{dx} &= \frac{d - cy - bz}{a}
\end{aligned}
\qquad
\begin{cases}
y(0) = y_0 \\
z(0) = y_1
\end{cases}
\tag{8.5}
$$

8.2   Explicit

8.2.1   Euler's method

The **Euler's Method** is an iterative method to solve ODEs. This method's huge plus is it is fast. The minus is it gradually diverges from the ideal solution.

The **Euler's Method** uses the following formula:

$$y(t + h) = y(t) + hf(x, y) \tag{8.6}$$

to construct the tangent at point x and obtain the value of $y(x+h)$, whose slope is:

$$f(x, y) \quad or \quad \frac{dy}{dx} \tag{8.7}$$

In Euler's method, you can approximate the curve of the solution byt the tangent in each interval (that is, by a sequence of shor line segments), at steps of h.

In general, if you use small step size, the accuracy of the approximation increases.
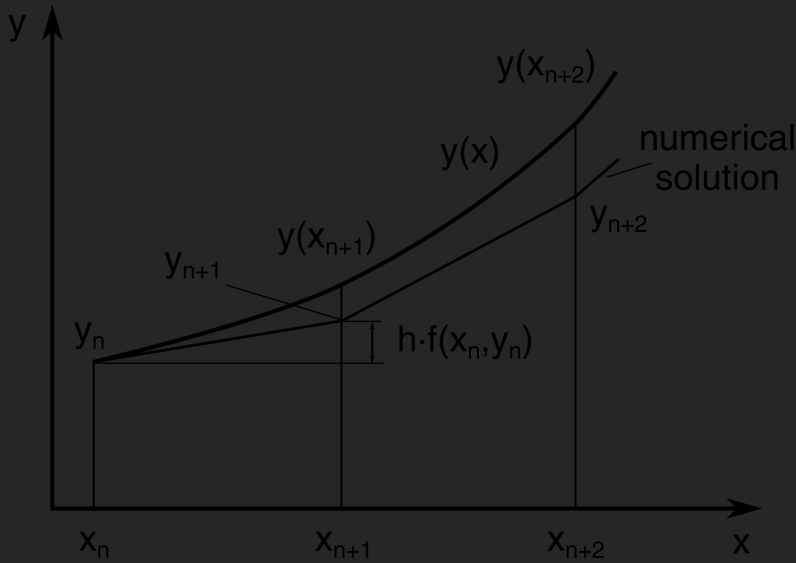
Figure 8.1: Euler's method schematic

General Formula

$$y_{i+1} = y_i + hf(x_i, y_i) \tag{8.8}$$

where:
y_{i+1} is the next estimated solution value,
y_i is the current value,
h is the interval between steps and
$f(x_i, y_i)$ is the value of the derivative at the current (x_i, y_i) point.

Pseudocode:

- define: $f(x, y)$
- input: x_0, y_0
- input: h, n
- for j from 0 to $(n - 1)$ do

- $y_{j+1} = y_j + hf(x_j, y_j)$
- $x_{j+1} = x_j + h$
- Print x_{j+1} and $= y_{j+1}$

- End.

Note:
If thinking about a problem in time domain:

- define: $f(t, x)$

- input: t_0, x_0

- input: dt, n

- for j from 0 to $(n-1)$ do

  - $x_{j+1} = x_j + dtf(t_j, x_j)$
  - $t_{j+1} = t_j + dt$
  - Print t_{j+1} and $= x_{j+1}$

- end.

Python Code for vibration problem:

```python
#!/usr/bin/python3
import numpy as np
import matplotlib.pyplot as plt


def iterate(h, y0, func, rhs):
    num_of_odes = y0.shape[0]
    y1 = np.zeros(num_of_odes, dtype = float)
    for i in range(num_of_odes-1):
        y1[i] = y0[i] + y0[i+1] * h
    y1[-1] = func(y1, rhs)
    return y1


def euler(t, y, func, rhs):
    N = t.shape[0]
    for j in range(N-1):
        dt = t[j+1] - t[j]
        y[j+1,:] = iterate(dt, y[j,:], func, rhs[j])
        # print(y[j+1,:])
    return y


def euler_vibration():
    m = 0.55  # tonnes
    c = 3.    # Ns/mm
    k = 1000. # N/mm

    freq = np.sqrt(k / m) / (2 * np.pi)

    # equation to solve
    # m * a + c * v + k * u = f
    # a = (f - c * v - k * u) / m
    acceleration = lambda u, f: (f - c * u[1] - k * u[0]) / m

    T = 2.0     # seconds
    dt = 0.0001 # timestep s
    u_0 = 0.    # mm of initial displacement
```

```
39        v_0 = 0.     # mm/s of initial velocity
40
41        # times at which to solve
42        t = np.linspace(0, T, int(T/dt) + 1)
43
44        # force vector
45        F = 5000. # N of max impulse
46        t0 = 0.1   # impulse start time
47        t1 = 0.2   # impuls end time
48        f = np.zeros(t.shape[0], dtype=float)
49        # create a half sine impulse of force
50        for i in range(t.shape[0]):
51            if t[i] >= t0 and t[i] <= t1:
52                f[i] = F * np.sin((t[i] - t0) / (t1 - t0) * np.pi)
53            else:
54                f[i] = 0.
55
56        uva = np.zeros((t.shape[0],3), dtype=float)
57        uva[0,0] = u_0
58        uva[0,1] = v_0
59
60        uva = euler(t, uva, acceleration, f)
61
62        fig, axf = plt.subplots()
63        fig.subplots_adjust(right=0.60)
64
65        p1, = axf.plot(t, f, label='force', color='violet')
66        axu = axf.twinx()
67        p2, = axu.plot(t, uva[:,0], label='displacement', color='red')
68        axv = axf.twinx()
69        axv.spines.right.set_position(('axes', 1.20))
70        p3, = axv.plot(t, uva[:,1], label='velocity', color='blue')
71        axa = axf.twinx()
72        axa.spines.right.set_position(('axes', 1.40))
73        p4, = axa.plot(t, uva[:,2], label='acceleration', color='green')
74
75        axf.set_xlabel('Time [s]')
76        axf.set_ylabel('Force [N]')
77        axu.set_ylabel('Displacement [mm]')
78        axv.set_ylabel('Velocity [mm/s]')
79        axa.set_ylabel('Acceleration [mm/s2]')
```

```
80
81      axf.legend(handles=[p1, p2, p3, p4])
82
83      plt.show()
84
85  if __name__ == '__main__':
86      euler_vibration()
87
```

Which results to:
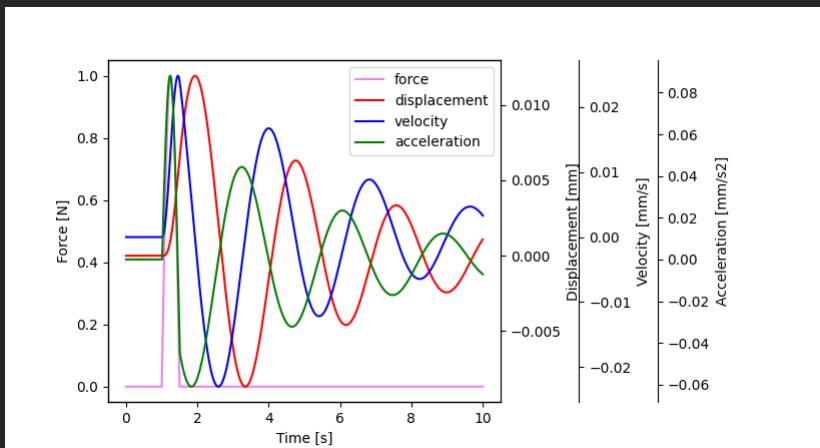
Figure 8.2: Euler's implementation for vibration problem example

The **iterate()** function:

```
1  def iterate(h, y0, func, rhs):
2      num_of_odes = y0.shape[0]
3      y1 = np.zeros(num_of_odes, dtype = float)
4      for i in range(num_of_odes-1):
5          y1[i] = y0[i] + y0[i+1] * h
6      y1[-1] = func(y1, rhs)
7      return y1
8
```

is written for general case of a set of ODEs. When considering the problem of vibration (set of 2 ODEs):

```python
def iterate(h, u0, v0, a0, f, m , c, k):
    u1 = u0 + h * v0
    v1 = v0 + h * a0
    # m * a + c * v + k * u = f
    a1 = (f - c * v1 - k * u1) / m
    return np.array([u1, v1, a1], dtype=float)
```

which is the iteration of a set of **first order ODEs**:

$$v = \dot{u}$$
$$\dot{v} = \frac{f - c * v - k * u}{m} \tag{8.9}$$

8.2.2 Runge-Kutta 4th order method

Runge-Kutta 4th order method is another explicit method for solving **first order ODEs**. The basic equation is:

$$\frac{dy}{dx} = f(x, y) \quad , y(0) = y_0 \tag{8.10}$$

The formula for the next value y_{i+1} after a step size equal to h is given by:

$$
\begin{aligned}
k_1 &= hf(x_i, y_i) \\
k_2 &= hf(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}) \\
k_3 &= hf(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}) \\
k_4 &= hf(x_i + h, y_i + k_3) \\
y_{i+1} &= y_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{aligned}
\tag{8.11}
$$

The formula basically computes next value y_{i+1} using current y_i plus **weighted average of four increments**:

- k_1 is the increment based on the slope at the beginning of the interval, using y

- k_2 is the increment based on the slope at the midpoint of the interval, using $y + hk_1/2$

- k_3 is the increment based on the slope at the midpoint, using $y + hk_2/2$

- k_4 is the increment based on the slope at the end of the interval, using $y + hk_3/2$

The method is a fourth order method, meaning that the local truncation error is on the order of $O(h^5)$, while the total accumulated error is of order $O(h^4)$.
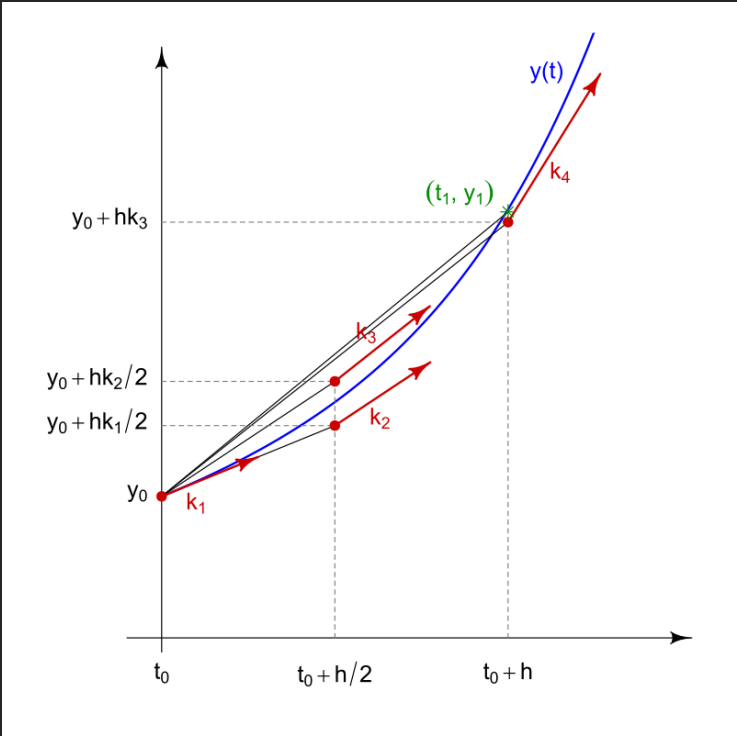
Figure 8.3: 4th Order Runge-Kutta's method schematic

Note:
The formula for the next value y_{i+1} can be also written as:

$$
\begin{aligned}
k_1 &= y_i \\
k_2 &= y_i + \frac{h}{2}\frac{dk_1}{dx} \\
k_3 &= y_i + \frac{h}{2}\frac{dk_2}{dx} \\
k_4 &= y_i + h\frac{k_3}{dx} \\
y_{i+1} &= y_i + \frac{h}{6}(\frac{dk_1}{dx} + 2\frac{dk_2}{dx} + 2\frac{dk_3}{dx} + \frac{dk_4}{dx}) + O(h^5)
\end{aligned}
\tag{8.12}
$$

This notation is more convenient to use when solving sets of ODEs for vibration problem, because it translates to:

$$u_1 = u_0$$
$$v_1 = v_0$$
$$a_1 = (f - cv_0 - ku_0)/m$$
$$u_2 = u_0 + \frac{h}{2}v_1$$
$$v_2 = v_0 + \frac{h}{2}a_1$$
$$a_2 = (f - cv_1 - ku_1)/m$$
$$u_3 = u_0 + \frac{h}{2}v_2$$
$$v_3 = v_0 + \frac{h}{2}a_2 \qquad (8.13)$$
$$a_3 = (f - cv_2 - ku_2)/m$$
$$u_4 = u_0 + hv_3$$
$$v_4 = v_0 + ha_3$$
$$a_4 = (f - cv_3 - ku_3)/m$$
$$u_{i+1} = u_0 + \frac{h}{6}(v_1 + 2v_2 + 2v_3 + v_4)$$
$$v_{i+1} = v_0 + \frac{h}{6}(a_1 + 2a_2 + 2a_3 + a_4)$$
$$a_{i+1} = (f - cv_{i+1} - ku_{i+1})/m$$

where each of the **ODEs** are solved sequentially from the values already known.

Python implementation for vibration problem:

```python
#!/usr/bin/python3
import numpy as np
import matplotlib.pyplot as plt


def iterate(h, y0, y, func, rhs):
    num_of_odes = y0.shape[0]
    y1 = np.zeros(num_of_odes, dtype = float)
    for i in range(num_of_odes-1):
        y1[i] = y0[i] + y[i+1] * h
    y1[-1] = func(y1, rhs)
    return y1


def runge_kutta_4(t, y, func, rhs):
    N = t.shape[0]
    for j in range(N-1):
        dt = t[j+1] - t[j]
        y1 = iterate(0., y[j,:], y[j,:], func, rhs[j])
        y2 = iterate(dt/2, y[j,:], y1, func, 0.5 * (rhs[j+1] + rhs[j]))
        y3 = iterate(dt/2, y[j,:], y2, func, 0.5 * (rhs[j+1] + rhs[j]))
        y4 = iterate(dt, y[j,:], y3, func, rhs[j+1])

        # next step solution
        for i in range(y.shape[1]-1):
            y[j+1,i] = y[j,i] + dt/6 * (y1[i+1] + 2 * y2[i+1] + 2 * y3[i+1]
                ↪   + y4[i+1])
        y[j+1,-1] = func(y[j+1,:], rhs[j+1])
    return y


def rk4_vibration():
    m = 10.    # tonnes
    c = 5.     # Ns/mm
    k = 50.    # N/mm

    freq = np.sqrt(k / m) / (2 * np.pi)
    print(f'f = {freq}')
```

```
38
39       # equation to solve
40       # m * a + c * v + k * u = f
41       # a = (f - c * v - k * u) / m
42       acceleration = lambda u, f: (f - c * u[1] - k * u[0]) / m
43
44       T = 10.      # seconds
45       dt = 0.01    # timestep s
46       u_0 = 0.     # mm of initial displacement
47       v_0 = 0.     # mm/s of initial velocity
48
49       # times at which to solve
50       t = np.linspace(0, T, int(T/dt) + 1)
51
52       # force vector
53       F = 1.     # N of max impulse
54       t0 = 1.    # impulse start time
55       t1 = 1.5   # impuls end time
56       f = np.zeros(t.shape[0], dtype=float)
57       # create a half sine impulse of force
58       for i in range(t.shape[0]):
59           if t[i] >= t0 and t[i] <= t1:
60               f[i] = F * np.sin((t[i] - t0) / (t1 - t0) * np.pi)
61           else:
62               f[i] = 0.
63
64       uva = np.zeros((t.shape[0],3), dtype=float)
65       uva[0,0] = u_0
66       uva[0,1] = v_0
67
68       uva = runge_kutta_4(t, uva, acceleration, f)
69
70       fig, axf = plt.subplots()
71       fig.subplots_adjust(right=0.60)
72
73       p1, = axf.plot(t, f, label='force', color='violet')
74       axu = axf.twinx()
75       p2, = axu.plot(t, uva[:,0], label='displacement', color='red')
76       axv = axf.twinx()
77       axv.spines.right.set_position(('axes', 1.20))
78       p3, = axv.plot(t, uva[:,1], label='velocity', color='blue')
```

```
79      axa = axf.twinx()
80      axa.spines.right.set_position(('axes', 1.40))
81      p4, = axa.plot(t, uva[:,2], label='acceleration', color='green')
82
83      axf.set_xlabel('Time [s]')
84      axf.set_ylabel('Force [N]')
85      axu.set_ylabel('Displacement [mm]')
86      axv.set_ylabel('Velocity [mm/s]')
87      axa.set_ylabel('Acceleration [mm/s2]')
88
89      axf.legend(handles=[p1, p2, p3, p4])
90
91      plt.show()
92
93  if __name__ == '__main__':
94      rk4_vibration()
95
```

Which results to:
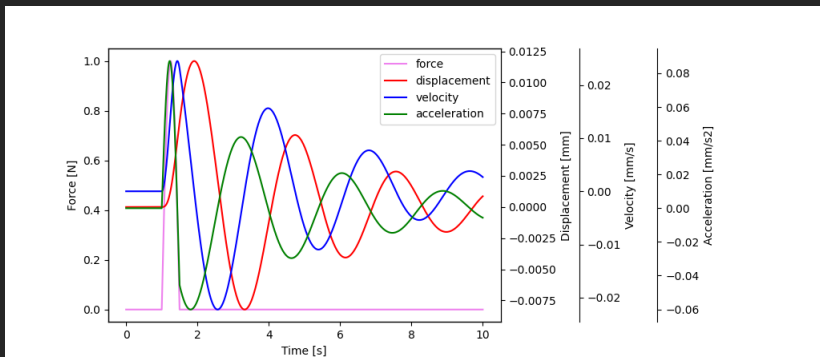
Figure 8.4: 4th order Runge-Kutta's implementation for vibration problem example

8.3   Implicit

The basic approach is an incremental step-by-step solution with an assumption that the solution for a discrete time t is known and that the solution for the discrete time $t + \Delta t$ is required, where Δt is a suitably chosen time increment. Hence, considering time $t + \Delta t$:

$$\mathbf{F}^{t+\Delta t} - \mathbf{N}^{t+\Delta t} = \mathbf{0} \tag{8.14}$$

Where $\mathbf{F}$ is the vector of external forces (loads) and $\mathbf{N}$ is the vector of inner forces.

Assume that $\mathbf{F}^{t+\Delta t}$ is independent of the deformations. Since the solution is known at time t, we can write:

$$\mathbf{N}^{t+\Delta t} = \mathbf{N}^t + \Delta \mathbf{N} \tag{8.15}$$

where $\Delta \mathbf{N}$ is the increment in nodal point forces corresponding to the increment in element displacements and stresses from time t to time $t + \Delta t$. This vector can be approximated using a **tangent stiffness matrix** $\mathbf{K}^t$ which corresponds to the geometric and material conditions at time t,

$$\Delta \mathbf{N} \doteq \mathbf{K}^t \Delta \mathbf{u} \tag{8.16}$$

where $\Delta \mathbf{u}$ is a vector of incremental nodal point displacements and

$$\mathbf{K}^t = \frac{\partial \mathbf{N}^t}{\partial \mathbf{u}^t} \tag{8.17}$$

Hence, the **tangent stiffness matrix** corresponds to the derivative of the internal element nodal point forces $\mathbf{N}^t$ with respect to the nodal point displacements $\mathbf{u}^t$.

Substituting (8.16) and (8.17) into (8.14), we obtain

$$\mathbf{K}^t \Delta \mathbf{u} = \mathbf{F}^{t+\Delta t} - \mathbf{N}^t \tag{8.18}$$

and solving for $\mathbf{u}^{\Delta t}$, we can calculate and approximation to the displacements at time $t + \Delta t$,

$$\mathbf{u}^{t+\Delta t} \doteq \mathbf{u}^t + \Delta \mathbf{u} \tag{8.19}$$

The exact displacements at time $t + \Delta t$ are those that correspond to the applied loads $\mathbf{F}^{t+\Delta t}$. We calculate in (8.19) only an approximation to these displacements because (8.16) was used.

Having evaluated an approximation to the displacements corresponding to time $t + \Delta t$, we could now solve for an approximation to the stresses and corresponding nodal point forces at time $t + \Delta t$, and then proceed to the next time increment calculations. However, because of the assumption (8.16), such a solution may be subject to very significant errors and, depending on the time or load step sizes used, may indeed be unstable. In practice, it is therefore necessary to iterate until the solution of (8.14) is obtained to sufficient accuracy.

8.3.1   Newton-Raphson

Having calculated an *increment* in nodal point displacements, which defones a *new total* displacement vector, we can repeat ht incremental solution using the currently known total displacements at time t.

The equations used in **Newton-Raphson** iteration are, *for* $i = 1, 2, 3, \ldots,$

$$\mathbf{K}_{i-1}^{t+\Delta t} \Delta \mathbf{u}_i = \mathbf{F}^{t+\Delta t} - \mathbf{N}_{i-1}^{t+\Delta t}$$
$$\mathbf{u}_i^{t+\Delta t} = \mathbf{u}_i^{t+\Delta t} + \mathbf{u}_{i-1}^{t+\Delta t} + \Delta \mathbf{u}_i \tag{8.20}$$

with the initial conditions:

$$\mathbf{u}_0^{t+\Delta t} = \mathbf{u}^t$$
$$\mathbf{K}_0^{t+\Delta t} = \mathbf{K}^t \tag{8.21}$$
$$\mathbf{N}_0^{t+\Delta t} = \mathbf{N}^t$$

Note that in the first iteration, the relations in (8.20) reduce to the equations (8.18) and (8.19). Then, in subsequent iterations, the latest estimates for the nodal point displacements are used to evaluate the corresponding element stresses and nodal point forces $\mathbf{N}_{i-1}^{t+\Delta t}$ and **tangent stiffness matrix** $\mathbf{K}_{i-1}^{t+\Delta t}$.

The out-of-balance load vector $\mathbf{F}^{t+\Delta t} - \mathbf{N}_{i-1}^{t+\Delta t}$ corresponds to a load vector that is not yet balanced by element stress, and hence an increment in the nodal point displacement is required. This updating of the nodal point displacement in the iteration is continued until out-of-balance loads and incremental displacements are small enough.

An important point is that the calculation of $\mathbf{N}_{i-1}^{t+\Delta t}$ from $\mathbf{u}_{i-1}^{t+\Delta t}$ is **crucial**. Any errors in this calculation will, in general, result in an incorrect response prediction.

The correct evaluation of the **tangent stiffness matrix** $\mathbf{K}_{i-1}^{t+\Delta t}$ is also important. The use of proper **tangent stiffness matrix** may be necessary for convergence and, in general, will result in fewer iterations until convergence is reached.

However, because the expense involved in evaluating and factoring a new **tangent stiffness matrix**, in practice, it can be more efficient, depending on the nonlinearities present in the analysis, to evaluate a new **tangent stiffness matrix** only at certain times. Specifically, in the **modified Newton-Raphson method** a new **tangent stiffness matrix** is established only at the beginning of each load step, and in **quasi-Newton** methods **secant stiffness matrices** are used instead of the tangent stiffness matrix.

The use of the iterative solution requires appropriate convergence criteria. If inappropriate criteria are used, the iteration may be terminated before the necessary solution accuracy is reached or be continued after the required accuracy has been reached.

Full Newton-Raphson Procedure derivation

The finite element equilibrium requirements amount to finding the solution of the equations:

$$f(\mathbf{u}^*) = 0 \qquad\qquad (8.22)$$

Where:

$$f(\mathbf{u}^*) = \mathbf{F}^{t+\Delta t}(\mathbf{u}^*) - \mathbf{N}^{t+\Delta t}(\mathbf{u}^*) \tag{8.23}$$

We denote here and in the following the complete array of the solution as $\mathbf{u}^*$ but realize that this vector may also contain variables other than displacements, for example, pressure variables and rotations.

Assume that in the iterative solution we have evaluated $\mathbf{u}_{i-1}^{t+\Delta t}$; then a Taylor series expansion gives:

$$f(\mathbf{u}^*) = f(\mathbf{u}_{i-1}^{t+\Delta t}) + \left[\frac{\partial N}{\partial \mathbf{u}}\right]\Bigg|_{\mathbf{u}_{i-1}^{t+\Delta t}} \left(\mathbf{u}^* - \mathbf{u}_{i-1}^{t+\Delta t}\right)$$
$$+ \text{ higher order terms} \tag{8.24}$$

Substituting from (8.23) into (8.24) and using (8.22), we obtain:

$$\left[\frac{\partial \mathbf{N}}{\partial \mathbf{u}}\right]\Bigg|_{\mathbf{u}_{i-1}^{t+\Delta t}} \left(\mathbf{u}^* - \mathbf{u}_{i-1}^{t+\Delta t}\right) + \text{ higher order terms}$$
$$= \mathbf{F}^{t+\Delta t} - \mathbf{N}_{i-1}^{t+\Delta t} \tag{8.25}$$

where we assumed that the externally applied loads are deformation independent.

Neglecting the higher-order terms in (8.25), we can calculate an increment in the displacements,

$$\mathbf{K}_{i-1}^{t+\Delta t}\Delta\mathbf{u}_i = \mathbf{F}^{t+\Delta t} - \mathbf{N}_{i-1}^{t+\Delta t} \tag{8.26}$$

where $\mathbf{K}_{i-1}^{t+\Delta t}$ is the current **tangent stiffness matrix**

$$\mathbf{K}_{i-1}^{t+\Delta t} = \left[\frac{\partial \mathbf{N}}{\partial \mathbf{u}}\right]\Bigg|_{\mathbf{u}_{i-1}^{t+\Delta t}} \tag{8.27}$$

and the improved displacement solution is:

$$\mathbf{u}_i^{t+\Delta t} = \mathbf{u}_{i-1}^{t+\Delta t} + \Delta \mathbf{u}_i \tag{8.28}$$

The relations in (8.26) and (8.28) cotitute the Newton-Raphson solution of (8.14). Since an incremental analysis is performed with time (or load) steps of size Δt, the initial conditions in this iteration are $\mathbf{K}_0^{t+\Delta t} = \mathbf{K}^t$, $\mathbf{N}_0^{t+\Delta t} = \mathbf{N}^t$, and $\mathbf{u}_0^{t+\Delta t} = \mathbf{u}^t$. The iteration is continued **until appropriate convergence criteria are satisfied.**

A characteristic of this iteration is that a new tangent stiffness matrix is calculated in **each** iteration, which is why this method is also referred to as the **full Newton-Raphson method**.
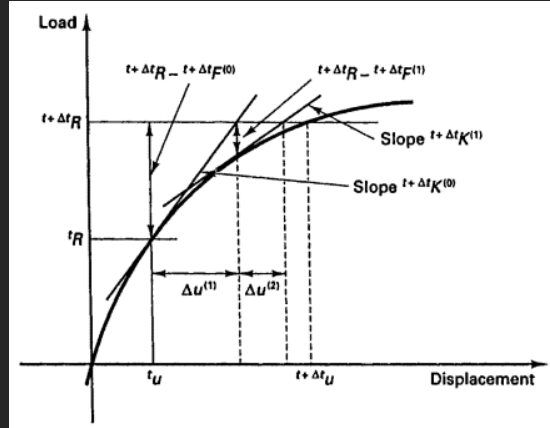
Figure 8.5: Newton-Raphson iteration in solution of SDOF system. **R** is load increment, **F** are internal forces, **u** are displacements.

Chapter 9

Model reduction

9.1   Guyan reduction

Guyan reduction, also known as **static condensation**, is a *dimensionality reduction* method which reduces the number of **DOFs** by ignoring the interial terms of the equilibrium equations and expressing the unloaded **DOFs** in terms of the loaded **DOFs**.

The static equilibrium equation:

$$
\begin{bmatrix} \mathbf{K}_{mm} & \mathbf{K}_{ms} \\ \mathbf{K}_{sm} & \mathbf{K}_{ss} \end{bmatrix} \begin{bmatrix} \mathbf{u}_m \\ \mathbf{u}_s \end{bmatrix} = \begin{bmatrix} \mathbf{f}_m \\ \mathbf{f}_s \end{bmatrix}
$$
$$
\mathbf{K}_{mm}\mathbf{u}_m + \mathbf{K}_{ms}\mathbf{u}_s = \mathbf{f}_m \tag{9.1}
$$
$$
\mathbf{K}_{sm}\mathbf{u}_m + \mathbf{K}_{ss}\mathbf{u}_s = \mathbf{f}_s
$$

Where m denotes the **master** DOFs and s the **slave** DOFs. Stating that s section of the problem DOFs is unloaded, we can write that $\mathbf{f}_s = \mathbf{0}$. Then the slave DOFs are expressed by the following equation:

$$
\mathbf{K}_{sm}\mathbf{u}_m + \mathbf{K}_{ss}\mathbf{u}_s = \mathbf{0} \tag{9.2}
$$

Solving the above equation in terms of the independent (master) DOFs leads to the following dependency relations:

$$\mathbf{u}_s = -\mathbf{K}_{ss}^{-1}\mathbf{K}_{sm}\mathbf{u}_m \tag{9.3}$$

Substituting the dependency relations to the upper (master) partition of the static equilibrium problem condenses away the slave DOFs, leading to the following reduced system of linear equations:

$$\left(\mathbf{K}_{mm} - \mathbf{K}_{ms}\mathbf{K}_{ss}^{-1}\mathbf{K}_{sm}\right)\mathbf{u}_m = \mathbf{f}_m \tag{9.4}$$

The above system of linear equations is equivalent to the original problem, but expressed in terms of **master** DOFs alone. Thus, the **Guyan** reduction method results in a reduced system by condensing away the **slave** DOFs.

The **Guyan reduction** can be also expressed as a *change of basis* which produces a low-dimensional representation of the original space, represented by the **master** DOFs. The linear transformation that maps the reduced space onto the full space is expressed as:

$$\begin{bmatrix} \mathbf{u}_m \\ \mathbf{u}_s \end{bmatrix} = \begin{bmatrix} \mathbf{I} \\ -\mathbf{K}_{ss}^{-1}\mathbf{K}_{sm} \end{bmatrix} \mathbf{u}_m = \mathbf{T}_G\mathbf{u}_m \tag{9.5}$$

where $\mathbf{T}_G$ represents the **Guyan** reduction *transformation matrix*. Thus, the reduced problem is represented as:

$$\mathbf{K}_G\mathbf{u}_m = \mathbf{f}_m \tag{9.6}$$

In the above equation, $\mathbf{K}_G$ represents the reduced system of linear equations that's obtained by applying the **Guyan reduction** transformation on the full system, which is expressed as:

$$\mathbf{K}_G = \mathbf{T}_G^T\mathbf{K}\mathbf{T}_G \tag{9.7}$$

where $\mathbf{T}_G$ has the dimension of $n_{DOF} \times n_m$.

Chapter 10

Element Atlas

10.1   Matrices

The inversion formula for 3x3 matrix:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\mathbf{A}^{-1} = \frac{1}{|\mathbf{A}|} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \tag{10.1}$$

where:

$$A_{11} = a_{22}a_{33} - a_{23}a_{32}$$
$$A_{22} = a_{33}a_{11} - a_{31}a_{13}$$
$$A_{33} = a_{11}a_{22} - a_{12}a_{21}$$
$$A_{12} = a_{23}a_{31} - a_{21}a_{33}$$
$$A_{23} = a_{31}a_{12} - a_{32}a_{11}$$
$$A_{31} = a_{12}a_{23} - a_{13}a_{22} \tag{10.2}$$
$$A_{21} = a_{32}a_{13} - a_{12}a_{33}$$
$$A_{32} = a_{13}a_{21} - a_{23}a_{11}$$
$$A_{13} = a_{21}a_{22} - a_{31}a_{22}$$
$$|A_{13}| = a_{11}A_{11} + a_{12}A_{21} + a_{13}A_{31}$$

10.2   Numerical Integration

> **Note:**
> **Quadrature** is another term for **Numerical Integration**.

10.2.1   Newton-Cotes quadrature

In the most obvious procedure, points at which the function is to be formed are determined *a priori* - usually at equal intervals - and a polynomial passed through the values of the function at these points and exactly integrated.

As n values define a polynomial of degree $n - 1$, the errors will be of the order $O(h^n)$ where h is the element size. This leads to the well-known **Newton-Cotes** 'quadrature formulae'. The integrals can be written as:

$$I = \int_{-1}^{1} f(\xi)d\xi = \sum_{1}^{n} H_i f(\xi_i) \tag{10.3}$$

for the range of integration between -1 and $+1$. For example, if $n = 2$, we have the well-known trapezoidal rule:

$$I = f(-1) + f(1) \tag{10.4}$$

for $n = 3$, the well-known **Simpson** one-third rule:

$$I = \frac{1}{3}\left[f(-1) + 4f(0) + f(1)\right] \tag{10.5}$$

and for $n = 4$:

$$I = \frac{1}{4}\left[f(-1) + 3f(-\frac{1}{3}) + 3f(\frac{1}{3}) + f(1)\right] \tag{10.6}$$

10.2.2   Gauss quadrature

If in place of specifying the position of sampling points *a priori* we allow these to be located at points to be determined so as to aim for best accuracy, then for a given number of sampling points increased accuracy can be obtained. Indeed, if we again consider:

$$I = \int_{-1}^{1} f(\xi)d\xi = \sum_{1}^{n} H_i f(\xi_i) \tag{10.7}$$

and again assume a polynomial expression, it is easy to see that for n sampling points we have $2n$ unknowns (H_i and ξ_i) and hence a polynomial of degree $2n - 1$ could be constructed and exactly integrated. The error is thus of order $O(h^{2n})$.

The simultaneous equations involved are difficult to solve, but some mathematical manipulation will show that the solution can be obtained explicitely in terms of **Legendre polynomials**. Thus this particular process is frequently known as **Gauss-Legendre** quadrature.

For purposes of finite elements analysis complex calculations are involved in determining the values of f, the function to be integratet. Thus the Gauss-type processes, requiring the least number of such evaluations, are ideally suited and are mostly used exvlusively.

Other expresssions for integration of functions of the type

$$I = \int_{-1}^{1} w(\xi) f(\xi) d\xi = \sum_{1}^{n} H_i f(\xi_i) \qquad (10.8)$$

can be derived for prescribed forms of $w(\xi)$, again integrating up to a certain order of accuracy a polynomial expansion of $f(\xi)$.

10.2.3   Linear and Quadrilateral Elements

For 1D:

$$I = \int_{-1}^{1} f(\xi) d\xi = \sum_{1}^{n} H_i f(\xi_i) \qquad (10.9)$$

For 2D:

The most obvious way of obtaining the integral:

$$I = \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta) d\xi d\eta \qquad (10.10)$$

Is to first evaluate the inner integral keeping η constant, i.e.:

$$\int_{-1}^{1} f(\xi, \eta) d\xi = \sum_{j=1}^{n} H_j f(\xi_i, \eta) = \psi(\eta) \qquad (10.11)$$

Evaluating the outer integral in a similar manner, we have:

$$\begin{aligned} I = \int_{-1}^{1} \psi(\eta) d\eta &= \sum_{i=1}^{n} H_i \psi(\eta_i) \\ &= \sum_{i=1}^{n} H_i \sum_{j=1}^{n} H_j f(\xi_i, \eta_i) \qquad (10.12) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} H_i H_j f(\xi_j, \eta_i) \end{aligned}$$

For 3D:

For a 3D element we have similarly:

$$
\begin{aligned}
I &= \int_{-1}^{1} \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta, \mu) d\xi d\eta d\mu \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} H_i H_j f(\xi_i, \eta_j, \mu_k)
\end{aligned}
\tag{10.13}
$$

In the above, the number of integrating points in each direction was assumed to be the same. Clearly this is not necessary and on occasion it may be an advantage to use different numbers in each direction of integration.
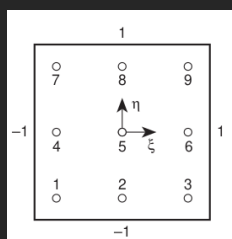
Figure 10.1: Integration points for $n = 3$ in a square region. Exact for a polynomial of fifth order in each direction

To get the **Gauss-Legendre** points and weights in python use:

```python
import numpy as np

# number of integration points
N = 3

points, weights = np.polynomial.legendre.leggauss(N)
```

integration point	polynomial order	weight
0	n = 1	2.0
$-1/\sqrt{3}$	n = 2	1.0
$1/\sqrt{3}$		1.0
$-\sqrt{0.6}$		5/9
0.0	n = 3	8/9
$\sqrt{0.6}$		5/9
-0.861 136 311 594 953		0.347 854 845 137 454
-0.339 981 043 584 856	n = 4	0.652 145 154 862 546
0.339 981 043 584 856		0.652 145 154 862 546
0.861 136 311 594 953		0.347 854 845 137 454
-0.906 179 845 938 664		0.236 926 885 056 189
-0.538 469 310 105 683		0.478 628 670 499 366
0.000 000 000 000 000	n = 5	0.568 888 888 888 889
0.538 469 310 105 683		0.478 628 670 499 366
0.906 179 845 938 664		0.236 926 885 056 189
-0.932 469 514 203 152		0.171 324 492 379 170
-0.661 209 386 466 265		0.360 761 573 048 139
-0.238 619 186 083 197	n = 6	0.467 913 934 572 691
0.238 619 186 083 197		0.467 913 934 572 691
0.661 209 386 466 265		0.360 761 573 048 139
0.932 469 514 203 152		0.171 324 492 379 170
-0.949 107 912 342 759		0.129 484 966 168 870
-0.741 531 185 599 394		0.279 705 391 489 277
-0.405 845 151 377 397		0.381 830 050 505 119
0.000 000 000 000 000	n = 7	0.471 959 183 673 469
0.405 845 151 377 397		0.381 830 050 505 119
0.741 531 185 599 394		0.279 705 391 489 277
0.949 107 912 342 759		0.129 484 966 168 870

Table 10.1: Gauss-Legendre Integration Points and Weights

10.2.4 Triangular and Tetrahedral elements:

2D

For a triangle, in terms of the **area coordinates** the integrals are of the form:

$$I = \int_0^1 \int_0^{1-\xi} f(\xi, \eta, \mu) d\eta d\xi, \quad \mu = 1 - \xi - \eta \qquad (10.14)$$

Once again we could use n Gauss points and arrive at a summation expression of the type used in the previous section. However, the limits of integration now involve the variable itself and it is convenient to use alternative sampling points for the second integration by use of a special Gauss expression for integrals of the type given by equation (10.8) in which w is a linear function. These have been devised by *Radau* and used successfully in finite element context. It is, however, much more desirable (and aesthetically pleasing) to use special formulae in which no bias is given to any of the natural coordinates (ξ, η, μ). Such formulae were first derived by *Hammer et al.* and *Felippa* and a series of necessary sampling points and weights.
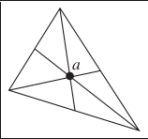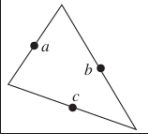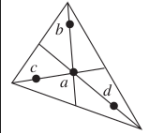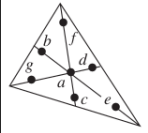
Order	Figure	Error	Points	Triangular Coordinates	Weights
Linear		$R = O(h^2)$	a	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$	1
Quadratic		$R = O(h^3)$	a b c	$\frac{1}{2}, \frac{1}{2}, 0$ $0, \frac{1}{2}, \frac{1}{2}$ $\frac{1}{2}, 0, \frac{1}{2}$	$\frac{1}{3}$ $\frac{1}{3}$ $\frac{1}{3}$
Cubic		$R = O(h^4)$	a b c d	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$ $0.6, 0.2, 0.2$ $0.2, 0.6, 0.2$ $0.2, 0.2, 0.6$	$-\frac{27}{48}$ $\frac{25}{48}$ $\frac{25}{48}$ $\frac{25}{48}$
Quintic		$R = O(h^6)$	a b c d e f g	$\frac{1}{3}, \frac{1}{3}, \frac{1}{3}$ $\alpha_1, \beta_1, \beta_1$ $\beta_1, \alpha_1, \beta_1$ $\beta_1, \beta_1, \alpha_1$ $\alpha_2, \beta_2, \beta_2$ $\beta_2, \alpha_2, \beta_2$ $\beta_2, \beta_2, \alpha_2$	0.2250000000 0.1323941527 0.1323941527 0.1323941527 0.1259391805 0.1259391805 0.1259391805

with:

$\alpha_1 = 0.059\ 715\ 871\ 7$

$\beta_1 = 0.470\ 142\ 064\ 1$

$\alpha_2 = 0.797\ 426\ 985\ 3$

$\beta_2 = 0.101\ 286\ 507\ 3$

Table 10.2: Triangular Element integration points and weights
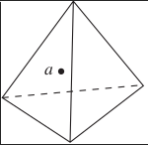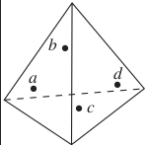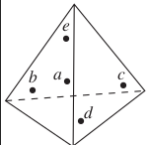
Order	Figure	Error	Points	Tetrahedral Coordinates	Weights
Linear		$R = O(h^2)$	a	$\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}$	1
Quadratic		$R = O(h^6)$	a	$\alpha, \beta, \beta, \beta$	$\frac{1}{4}$
			b	$\beta, \alpha, \beta, \beta$	$\frac{1}{4}$
			c	$\beta, \beta, \alpha, \beta$	$\frac{1}{4}$
			d	$\beta, \beta, \beta, \beta$	$\frac{1}{4}$
			with:		
			$\alpha = 0.585\ 410\ 20$		
			$\beta = 0.138\ 196\ 60$		
Cubic		$R = O(h^4)$	a	$\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}$	$-\frac{4}{5}$
			b	$\frac{1}{2}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}$	$\frac{9}{20}$
			c	$\frac{1}{6}, \frac{1}{2}, \frac{1}{6}, \frac{1}{6}$	$\frac{9}{20}$
			d	$\frac{1}{6}, \frac{1}{6}, \frac{1}{2}, \frac{1}{6}$	$\frac{9}{20}$
			e	$\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{2}$	$\frac{9}{20}$

Table 10.3: Tetrahedral Element integration points and weights

10.3   General Element Matrices Procedure

The procedure to generate any element stiffness, mass and load matrices
can be generalised to the following steps:

(a) Create Element domain in natural coordinates:

  - -1 to 1 for quadrilateral elements
  - 0 to 1 for triangular and linear elements

(b) select the number of Gauss points based on the order of the element

  - 1 or 2 for linear elements
  - 3 for quadratic elements

Example:

```
1  import numpy as np
2  gp, gw = np.polynomial.legendre.leggauss(3)
```

(c) Generate the shape functions $\mathbf{N}_n^e$ in natural coordinates based on the
number of Gauss points, dimension of the element and its domain.
The $\mathbf{N}_n^e$ matrix has the dimensions of:

  - number of integration points = number of rows
  - number of shape functions = number of columns

Therefore:

$$\mathbf{N}_n^e = \begin{bmatrix} N_{1,1}^e & \cdots & N_{n,1}^e \\ \vdots & \vdots & \vdots \\ N_{1,m}^e & \cdots & N_{n,m}^e \end{bmatrix} \tag{10.15}$$

where $1 \ldots m$ are the integration points and $1 \ldots n$ are the shape
functions.

(d) Transform the shape functions from natural coordinates to global
coordinates

$$\mathbf{N}_g^e = \mathbf{N}_n^e \mathbf{x}^e$$

$$\mathbf{N}_g^e = \begin{bmatrix} N_{1,1}^e(\xi) & \cdots & N_{n,1}^e(\xi) \\ \vdots & \vdots & \vdots \\ N_{1,m}^e(\xi) & \cdots & N_{n,m}^e(\xi) \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix} \tag{10.16}$$

The $\mathbf{x}^e$ matrix is a matrix of global coordinates of the element nodes. The resulting matrix has the dimension of:

- number of integration points = number of rows
- number of coordinates = number of columns (3D = 3, 2D = 2)

(e) Generate the shape functions derivatives $\mathbf{B}_n^e$ in natural coordinates

The shape functions derivatives matrix has 3 dimensions being *number of integration points $\times$ number of natural coordinates $\times$ number of shape functions*

(f) Create the Matrix of Jacobians from the shape functions derivatives:

$$\mathbf{J} = \mathbf{B}_n^e \mathbf{x}^e \tag{10.17}$$

The Matrix of Jacobians has the dimension of *number of integration points $\times$ number of global coordinates $\times$ number of global coordinates*, where number of global coordinates means 2 for 2D and 3 for 3D problem.

(g) Get the Matrix of Jacobian determinants and an inverse Jacobian:

$$\mathbf{J}^{-1}\mathbf{J}_d = det\mathbf{J} \tag{10.18}$$

Example:

```
1 d_jacobi = np.linalg.det(jacobi)
2 i_jacobi = np.linalg.inv(jacobi)
```

The Jacobian determinant is computed for each integration point and has a dimension of *number of integration points $\times$ 1*

The inverse Jacobian has the same dimension as the Jacobian (**must have**)

(h) Transform the shape function derviatives from natural coordinates to global coordinates

$$\mathbf{B}_g^e = \mathbf{J}^{-1}\mathbf{B}_n^e \tag{10.19}$$

(i) Create the Material Stiffness matrix $\mathbf{C}$

(j) Finally for each integration point:

3D:

$$\begin{aligned}
\mathbf{B}_{i,x}^e &= \mathbf{B}_g(i,0) \\
\mathbf{B}_{i,y}^e &= \mathbf{B}_g(i,1) \\
\mathbf{B}_{i,z}^e &= \mathbf{B}_g(i,2)
\end{aligned}$$

(10.20)

where $\mathbf{B}_{i,x}^e$, $\mathbf{B}_{i,y}^e$ and $\mathbf{B}_{i,z}^e$ are vectors of legth $=$ *number of shape functions*

Then matrix $\mathbf{B}$ is:

$$\mathbf{B}_i = \begin{bmatrix}
\mathbf{B}_{i,x}^e & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{B}_{i,y}^e & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{B}_{i,z}^e \\
\mathbf{B}_{i,y}^e & \mathbf{B}_{i,x}^e & \mathbf{0} \\
\mathbf{0} & \mathbf{B}_{i,z}^e & \mathbf{B}_{i,y}^e \\
\mathbf{B}_{i,z}^e & \mathbf{0} & \mathbf{B}_{i,x}^e
\end{bmatrix}$$

(10.21)

and matrix $\mathbf{N}$ is:

$$\mathbf{N}_i = \begin{bmatrix}
\mathbf{N}_{n,i}^e & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{N}_{n,i}^e & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{N}_{n,i}^e
\end{bmatrix}$$

(10.22)

where $\mathbf{0}$ is a zero vector of length $=$ *number of shape functions* and $\mathbf{N}_{n,i}^e$ is a vector of shape functions in natural coordinates pertaining to the i-th integration point.

Then:

  i.
$$\mathbf{K}_i^e = \mathbf{B}_i^T \mathbf{C} \mathbf{B}_i \; det\mathbf{J}_i w_i$$
(10.23)

  ii.
$$\mathbf{M}_i^e = \rho \mathbf{N}_i^T \mathbf{N}_i \; det\mathbf{J}_i w_i$$
(10.24)

  iii.
$$\mathbf{F}_i^e = \mathbf{N}_i^T \mathbf{F} \; det\mathbf{J}_i w_i$$
(10.25)

where $\mathbf{F} = \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix}$

where w_i is the multiple of gaussian weights for the respective integration point (for 3D case brick where the natural coordinates are ξ, η, μ the $w_i = w_{\xi_i} w_{\eta_i} w_{\mu_i}$

(k) Finally:

i.

$$\mathbf{K}^e = \sum_{i=1}^{n} \mathbf{K}_i^e \qquad (10.26)$$

ii.

$$\mathbf{M}^e = \sum_{i=1}^{n} \mathbf{M}_i^e \qquad (10.27)$$

iii.

$$\mathbf{F}^e = \sum_{i=1}^{n} \mathbf{F}_i^e \qquad (10.28)$$

10.4   ROD

10.4.1   Element Formulation

The **rod** element is a 1-D linear element. It has 2 nodes, and 3 dofs for each node u_x, u_y, u_y (or u, v, w). Rotation **DOF**s are not constrained.

Natural coordinates

The natural coordinate ξ of the **ROD** element goes from 0 to 1. The 2-noded rod element is defined by:

$$
\begin{bmatrix} 1 \\ x \\ y \\ z \\ u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ x_1 & x_2 \\ y_1 & y_2 \\ z_1 & z_2 \\ u_{x1} & u_{x2} \\ u_{y1} & u_{y2} \\ u_{z1} & u_{z2} \end{bmatrix} \begin{bmatrix} N_1^e \\ N_2^e \end{bmatrix} \tag{10.29}
$$

Shape Functions

The **ROD** natural coordinates are:

node	ξ
1	0
2	1

Table 10.4: ROD corners natural coordinates

The shape functions are:

$$
\begin{aligned}
N_1^e &= 1 - \xi \\
N_2^e &= \xi
\end{aligned}
\tag{10.30}
$$

Partial Derivatives

The shape function derivatives with respect to the natural coordinate:

$$
\begin{aligned}
\frac{\partial N_1^e}{\partial \xi} &= -1 \\
\frac{\partial N_2^e}{\partial \xi} &= 1
\end{aligned}
\tag{10.31}
$$

The Jacobian

The derivatives of the shape functions are given by the usual chain rule formula:

$$
\begin{aligned}
\frac{\partial N_i^e}{\partial x} &= \frac{\partial N_i^e}{\partial \xi}\frac{\partial \xi}{\partial x} \\
\frac{\partial N_i^e}{\partial y} &= \frac{\partial N_i^e}{\partial \xi}\frac{\partial \xi}{\partial y} \\
\frac{\partial N_i^e}{\partial z} &= \frac{\partial N_i^e}{\partial \xi}\frac{\partial \xi}{\partial z}
\end{aligned}
\tag{10.32}
$$

In matrix form:

$$
\begin{bmatrix}
\frac{\partial N_i^e}{\partial x} \\
\frac{\partial N_i^e}{\partial y} \\
\frac{\partial N_i^e}{\partial z}
\end{bmatrix}
=
\begin{bmatrix}
\frac{\partial \xi}{\partial x} \\
\frac{\partial \xi}{\partial y} \\
\frac{\partial \xi}{\partial z}
\end{bmatrix}
\begin{bmatrix}
\frac{\partial N_i^e}{\partial \xi}
\end{bmatrix}
\tag{10.33}
$$

The 3×1 matrix above is $\mathbf{J}^{-1}$, the inverse of:

$$
\mathbf{J} = \frac{\partial(x,y,z)}{\partial(\xi)} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \end{bmatrix}
\tag{10.34}
$$

Given the element coordinates the Jacobian can be computed:

$$
\mathbf{J}_i = \begin{bmatrix} \frac{\partial N_i^e}{\partial \xi} & \frac{\partial N_i^e}{\partial \xi} & \frac{\partial N_i^e}{\partial \xi} \end{bmatrix}
\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}
\tag{10.35}
$$

end then numerically inverted for $\mathbf{J}^{-1}$.

If the $\mathbf{J}$ is not a square matrix, then **Moore-Penrose pseudoinverse** applies:

$$\mathbf{J}^{-1} = \mathbf{J}^{+} = \left(\mathbf{J}^T \mathbf{J}\right)^{-1} \mathbf{J}^T \tag{10.36}$$

in python:

```python
1        import numpy as np
2
3        J_i = np.linalg.pinv(J)
```

Assembling the **Jacobian** thegether gives essentially that for a 1D case

$$\mathbf{J} = l \tag{10.37}$$

and therefore:

$$\mathbf{J}^{-1} = \frac{1}{l} \tag{10.38}$$

$$\det J = l \tag{10.39}$$

afterwards for each integration point:

$$\mathbf{N}_g^i = \mathbf{N}^i \mathbf{u} \tag{10.40}$$

$$\mathbf{B}_g^i = \mathbf{J}^{i-1} \frac{\partial N^i}{\partial \xi} \tag{10.41}$$

then:

$$\mathbf{K}^e \overset{+}{=} \mathbf{B}_g^{i\,T} \mathbf{D} \mathbf{B}_g^i \det \mathbf{J}^i w^i$$
$$\mathbf{M}^e \overset{+}{=} \rho \mathbf{N}^{i\,T} \mathbf{N}^i \det \mathbf{J}^i w^i \tag{10.42}$$
$$\mathbf{F}^e \overset{+}{=} \mathbf{N}^{i\,T} \mathbf{F} \det \mathbf{J}^i w^i$$

10.4.2 Example 1D

If assembling the element in local CSYS, then $u_y = 0$ and $u_z = 0$. Simplifiying the equations gives that:

$$\mathbf{N} = \begin{bmatrix} N_1^e & N_2^e \end{bmatrix} = \begin{bmatrix} 1 - \xi & \xi \end{bmatrix} \tag{10.43}$$

and

$$u^e(\xi) = \mathbf{N} \begin{bmatrix} u_1^e \\ u_2^e \end{bmatrix} \tag{10.44}$$

shape function derivatives:

$$\mathbf{B} = \frac{\partial \mathbf{N}}{\partial \xi} = \begin{bmatrix} -1 & 1 \end{bmatrix} \tag{10.45}$$

the strain-displacement realtion is given by:

$$\epsilon = \mathbf{Bu} \tag{10.46}$$

Integrating the element at midpoint $\xi = 0.5$ with a weight of $w = 1.0$ (for interval $\langle 0, 1 \rangle$, for an interval $\langle -1, 1 \rangle$ the weight is $w = 2.0$)

$$\mathbf{N} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \tag{10.47}$$

Shape Functions in global coordinates:

$$\mathbf{N} = 0.5u_1 + 0.5u_2 = 0.5(u_1 + u_2) \tag{10.48}$$

Shape Function Derivatives in Natural Coordinates:

$$\mathbf{B} = \begin{bmatrix} -1 & 1 \end{bmatrix} \tag{10.49}$$

Jacobian:

$$\mathbf{J} = \mathbf{Bu} = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = u_2 - u_1 = l \tag{10.50}$$

Jacobian Determinant:

$$\det \mathbf{J} = l \tag{10.51}$$

Jacobian Inverse:

$$\mathbf{J}^{-1} = \frac{1}{l} \tag{10.52}$$

Shape function derivatives in global coordinates:

$$\mathbf{B}_g = \mathbf{J}^{-1}\mathbf{B} = \frac{1}{l}\begin{bmatrix} -1 & 1 \end{bmatrix} \tag{10.53}$$

Material Stiffness Matrix

$$\mathbf{D} = EA \tag{10.54}$$

Then for each integration point (here 1):

$$\begin{aligned}
\mathbf{K}^e &\stackrel{\pm}{=} \mathbf{B}_g^T \mathbf{D} \mathbf{B_g} \det \mathbf{J} w \\
\mathbf{K}^e &\stackrel{\pm}{=} \frac{1}{l}\begin{bmatrix} -1 \\ 1 \end{bmatrix} EA \frac{1}{l}\begin{bmatrix} -1 & 1 \end{bmatrix} l 1 \\
\mathbf{K}^e &\stackrel{\pm}{=} \frac{EA}{l}\begin{bmatrix} -1 \\ 1 \end{bmatrix}\begin{bmatrix} -1 & 1 \end{bmatrix} \\
\mathbf{K}^e &\stackrel{\pm}{=} \frac{EA}{l}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}
\end{aligned} \tag{10.55}$$

10.4.3   Example 2D

If assembling the element in 2D, then $u_z = 0$. Simplifiying the equations gives that:

$$\mathbf{N} = \begin{bmatrix} N_1^e & 0 & N_2^e & 0 \\ 0 & N_1^e & 0 & N_2^e \end{bmatrix} = \begin{bmatrix} 1-\xi & 0 & \xi & 0 \\ 0 & 1-\xi & 0 & \xi \end{bmatrix} \tag{10.56}$$

and

$$\begin{bmatrix} u_x^e(\xi) \\ u_y^e(\xi) \end{bmatrix} = \mathbf{N} \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{x2}^e \\ u_{y2}^e \end{bmatrix} \tag{10.57}$$

or if:

$$\mathbf{N} = \begin{bmatrix} N_1^e & N_2^e \end{bmatrix} = \begin{bmatrix} 1 - \xi & \xi \end{bmatrix} \tag{10.58}$$

then:

$$\begin{bmatrix} u_x^e(\xi) \\ u_y^e(\xi) \end{bmatrix} = \mathbf{N} \begin{bmatrix} u_{x1}^e & u_{y1}^e \\ u_{x2}^e & u_{y2}^e \end{bmatrix} \tag{10.59}$$

shape function derivatives:

$$\mathbf{B} = \frac{\partial \mathbf{N}}{\partial \xi} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{10.60}$$

the strain-displacement realtion is given by:

$$\boldsymbol{\epsilon} = \mathbf{Bu} \tag{10.61}$$

Integrating the element at midpoint $\xi = 0.5$ with a weight of $w = 1.0$ (for interval $\langle 0, 1 \rangle$, for an interval $\langle -1, 1 \rangle$ the weight $w = 2.0$) then numerically:

$$\mathbf{N} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \tag{10.62}$$

Shape Functions in global coordinates:

$$\mathbf{N} = \begin{bmatrix} 1 - \xi & 0 & \xi & 0 \\ 0 & 1 - \xi & 0 & \xi \end{bmatrix}_{\xi = 0.5} \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{x2}^e \\ u_{y2}^e \end{bmatrix} = \begin{bmatrix} 0.5u_{x1} + 0.5u_{x2} \\ 0.5u_{y1} + 0.5u_{y2} \end{bmatrix} \tag{10.63}$$

Shape Function Derivatives in Natural Coordinates (there is a row for each integration point if the coordinates are specified as matrix, not a column vector. otherwise for each integration point there is a separate $\mathbf{B}$ matrix):

$$\frac{\partial \mathbf{N}}{\partial \xi} = \mathbf{B} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{10.64}$$

Jacobian:

$$\mathbf{J} = \mathbf{Bu} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{x2}^e \\ u_{y2}^e \end{bmatrix} = \begin{bmatrix} u_{x2}^e - u_{x1}^e \\ u_{y2}^e - u_{y1}^e \end{bmatrix} = \begin{bmatrix} l_x \\ l_y \end{bmatrix} \tag{10.65}$$

? ?

Jacobian Determinant (when the **Jacobian matrix** is a vector, its deter-
minant is simply its **length**):

¿ ¿

$$\det \mathbf{J} = ||\mathbf{J}|| = \sqrt{J_1^2 + J_2^2} = \sqrt{l_x^2 + l_y^2} = l \qquad (10.66)$$

? ?

Jacobian Inverse (when the **Jacobian matrix** is a row vector, its inverse
is just its **transpose** divided by its determinant squared):

Moore-Penrose pseudoinverse:

$$\begin{aligned}
\boxed{\mathbf{J}^{-1}} = \mathbf{J}^+ &= \left(\mathbf{J}^T \mathbf{J}\right)^{-1} \mathbf{J}^T \\
&= \left(\begin{bmatrix} l_x & l_y \end{bmatrix} \begin{bmatrix} l_x \\ l_y \end{bmatrix} \right)^{-1} \begin{bmatrix} l_x & l_y \end{bmatrix} \\
&= \left(l_x^2 + l_y^2\right)^{-1} \begin{bmatrix} l_x & l_y \end{bmatrix} \\
&= \frac{1}{l_x^2 + l_y^2} \begin{bmatrix} l_x & l_y \end{bmatrix} \qquad (10.67) \\
&= \frac{1}{\sqrt{l_x^2 + l_y^2}^2} \begin{bmatrix} l_x & l_y \end{bmatrix} \\
&= \boxed{\frac{1}{l^2} \mathbf{J}^T}
\end{aligned}$$

¿ ¿

$$\mathbf{J}^{-1} = \frac{1}{l^2} \begin{bmatrix} l_x & l_y \end{bmatrix} = \frac{1}{l} \begin{bmatrix} c & s \end{bmatrix} \qquad (10.68)$$

where $c = \cos\varphi = l_x/l$ and $s = \sin\varphi = l_y/l$.

Shape function derivatives in global coordinates:

$$\mathbf{B}_g = \mathbf{J}^{-1}\mathbf{B} = \frac{1}{l}\begin{bmatrix} c & s \end{bmatrix}\begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} = \frac{1}{l}\begin{bmatrix} -c & -s & c & s \end{bmatrix} \quad (10.69)$$

Material Stiffness Matrix

$$\mathbf{D} = EA \qquad (10.70)$$

Then for each integration point (here 1):

$$\mathbf{K}^e \pm \mathbf{B}_g^T \mathbf{D} \mathbf{B_g} \det \mathbf{J} w \qquad (10.71)$$

$$\mathbf{K}^e \pm \frac{1}{l}\begin{bmatrix} -c \\ -s \\ c \\ s \end{bmatrix} EA\frac{1}{l}\begin{bmatrix} -c & -s & c & s \end{bmatrix} lw$$

$$\pm \frac{EA}{l}\begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix} w \qquad (10.72)$$

which is the same result as when the element stiffness matrix is derived in local coordinate system for a situation where ξ axis coincides with the element x axis and the u_y component is 0. Afterwards, the element **stiffness** and **mass** matrices as well as **forces** vector are tranformed to the global CSYS using a transformation matrix $\mathbf{T}$:

$$\mathbf{K}_l^e = \frac{EA}{l}\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad (10.73)$$

$$\mathbf{T} = \begin{bmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & c & s \\ 0 & 0 & -s & c \end{bmatrix} \qquad (10.74)$$

then the global element stiffness matrix is defined:

$$\mathbf{K}_g^e = \mathbf{T}^T \mathbf{K}_l^e \mathbf{T}$$

$$= \frac{EA}{l} \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & c & s \\ 0 & 0 & -s & c \end{bmatrix}$$

$$= \frac{EA}{l} \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & c & -s \\ 0 & 0 & s & c \end{bmatrix} \begin{bmatrix} c & s & -c & -s \\ 0 & 0 & 0 & 0 \\ -c & -s & c & s \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{10.75}$$

$$= \frac{EA}{l} \begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix}$$

10.4.4   Example 3D

If assembling the element in 3D, then $u_x \neq 0$, $u_y \neq 0$ and $u_z \neq 0$. The full equations give that:

$$\mathbf{N} = \begin{bmatrix} N_1^e & 0 & 0 & N_2^e & 0 & 0 \\ 0 & N_1^e & 0 & 0 & N_2^e & 0 \\ 0 & 0 & N_1^e & 0 & 0 & N_2^e \end{bmatrix} = \begin{bmatrix} 1-\xi & 0 & 0 & \xi & 0 & 0 \\ 0 & 1-\xi & 0 & 0 & \xi & 0 \\ 0 & 0 & 1-\xi & 0 & 0 & \xi \end{bmatrix} \tag{10.76}$$

and

$$\begin{bmatrix} u_x^e(\xi) \\ u_y^e(\xi) \\ u_z^e(\xi) \end{bmatrix} = \mathbf{N}(\xi) \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{z1}^e \\ u_{x2}^e \\ u_{y2}^e \\ u_{z2}^e \end{bmatrix} \tag{10.77}$$

or if:

$$\mathbf{N} = \begin{bmatrix} N_1^e & N_2^e \end{bmatrix} = \begin{bmatrix} 1-\xi & \xi \end{bmatrix} \tag{10.78}$$

then:

$$\begin{bmatrix} u_x^e(\xi) \\ u_y^e(\xi) \\ u_z^e(\xi) \end{bmatrix} = \mathbf{N}(\xi) \begin{bmatrix} u_{x1}^e & u_{y1}^e & u_{z1}^e \\ u_{x2}^e & u_{y2}^e & u_{z2}^e \end{bmatrix} \tag{10.79}$$

shape function derivatives:

$$\mathbf{B} = \frac{\partial \mathbf{N}}{\partial \xi} = \begin{bmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \tag{10.80}$$

the strain-displacement relation is given by:

$$\epsilon = \mathbf{B}\mathbf{u} \tag{10.81}$$

Integrating the element at midpoint $\xi = 0.5$ with a weight of $w = 1.0$ (for interval $\langle 0, 1 \rangle$, for an interval $\langle -1, 1 \rangle$ the weight $w = 2.0$) then numerically:

$$\mathbf{N} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \tag{10.82}$$

Shape Functions in global coordinates:

$$\mathbf{N} = \begin{bmatrix} 1-\xi & 0 & 0 & \xi & 0 & 0 \\ 0 & 1-\xi & 0 & 0 & \xi & 0 \\ 0 & 0 & 1-\xi & 0 & 0 & \xi \end{bmatrix}_{\xi=0.5} \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{z1}^e \\ u_{x2}^e \\ u_{y2}^e \\ u_{z2}^e \end{bmatrix} = \begin{bmatrix} 0.5u_{x1} + 0.5u_{x2} \\ 0.5u_{y1} + 0.5u_{y2} \\ 0.5u_{z1} + 0.5u_{z2} \end{bmatrix} \tag{10.83}$$

Shape Function Derivatives in Natural Coordinates (there is a row for each integration point if the coordinates are specified as matrix, not a column vector. otherwise for each integration point there is a separate $\mathbf{B}$ matrix):

$$\frac{\partial \mathbf{N}}{\partial \xi} = \mathbf{B} = \begin{bmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \tag{10.84}$$

Jacobian:

$$\mathbf{J} = \mathbf{Bu} = \begin{bmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix}_{\xi=0.5} \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{z1}^e \\ u_{x2}^e \\ u_{y2}^e \\ u_{z2}^e \end{bmatrix} = \begin{bmatrix} u_{x2}^e - u_{x1}^e \\ u_{y2}^e - u_{y1}^e \\ u_{z2}^e - u_{z1}^e \end{bmatrix} = \begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix}$$

$$(10.85)$$

? ?

Jacobian Determinant (when the **Jacobian matrix** is a vector, its determinant is simply its **length**):

¿ ¿

$$\det \mathbf{J} = ||\mathbf{J}|| = \sqrt{J_1^2 + J_2^2 + J_3^2} = \sqrt{l_x^2 + l_y^2 + l_y^2} = l \qquad (10.86)$$

? ?

Jacobian Inverse (when the **Jacobian matrix** is a row vector, its inverse is just its **transpose** divided by its determinant squared):

Moore-Penrose pseudoinverse:

$$\begin{aligned} \boxed{\mathbf{J}^{-1}} &= \mathbf{J}^+ = \left(\mathbf{J}^T \mathbf{J}\right)^{-1} \mathbf{J}^T \\ &= \left(\begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix} \right)^{-1} \begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \\ &= \left(l_x^2 + l_y^2 + l_z^2 \right)^{-1} \begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \\ &= \frac{1}{l_x^2 + l_y^2 + l_z^2} \begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \\ &= \frac{1}{\sqrt{l_x^2 + l_y^2 + l_z^2}^2} \begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \\ &= \boxed{\frac{1}{l^2} \mathbf{J}^T} \end{aligned} \qquad (10.87)$$

¿ ¿

$$\mathbf{J}^{-1} = \frac{1}{l^2} \begin{bmatrix} l_x & l_y & l_z \end{bmatrix} \tag{10.88}$$

Shape function derivatives in global coordinates:

$$\mathbf{B}_g = \mathbf{J}^{-1}\mathbf{B} = \frac{1}{l^2} \begin{bmatrix} l_x & l_y & l_z \end{bmatrix}_{\xi=0.5} \begin{bmatrix} -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix}_{\xi=0.5} \tag{10.89}$$

$$= \frac{1}{l^2} \begin{bmatrix} -l_x & -l_y & -l_z & l_x & l_y & l_z \end{bmatrix}$$

Material Stiffness Matrix

$$\mathbf{D} = EA \tag{10.90}$$

Then for each integration point (here 1):

$$\mathbf{K}^e \stackrel{\pm}{=} \mathbf{B}_g^T \mathbf{D} \mathbf{B_g} \det \mathbf{J} w \tag{10.91}$$

$$
\mathbf{K}^e \stackrel{\pm}{=} \frac{1}{l^2} \begin{bmatrix} -l_x \\ -l_y \\ -l_z \\ l_x \\ l_y \\ l_z \end{bmatrix} EA \frac{1}{l^2} \begin{bmatrix} -l_x & -l_y & -l_z & l_x & l_y & l_z \end{bmatrix} lw
$$

$$
\stackrel{\pm}{=} \frac{EA}{l^3} \begin{bmatrix}
l_x^2 & l_x l_y & l_x l_z & -l_x^2 & -l_x l_y & -l_x l_z \\
l_x l_y & l_y^2 & l_y l_z & -l_x l_y & -l_y^2 & -l_y l_z \\
l_x l_z & l_y l_z & l_z^2 & -l_x l_z & -l_y l_z & -l_z^2 \\
-l_x^2 & -l_x l_y & -l_x l_z & l_x^2 & l_x l_y & l_x l_z \\
-l_x l_y & -l_y^2 & -l_y l_z & l_x l_y & l_y^2 & l_y l_z \\
-l_x l_z & -l_y l_z & -l_z^2 & l_x l_z & l_y l_z & l_z^2
\end{bmatrix} w
\tag{10.92}
$$

10.4.5   Example 2D quadratic

If assembling the element in 2D, then and $u_z = 0$.

The node numbering scheme is $1 - 3 - 2$ and the natural coordinate is defined $\xi \in \langle -1, 1 \rangle$.

The **quadratic** shape functions are:

$$N_1^e = -\frac{1}{2}\xi\left(1 - \xi\right)$$
$$N_2^e = \frac{1}{2}\xi\left(1 + \xi\right) \tag{10.93}$$
$$N_3^e = \left(1 + \xi\right)\left(1 - \xi\right)$$

$$\mathbf{N} = \begin{bmatrix} N_1^e & 0 & N_2^e & 0 & N_3^e & 0 \\ 0 & N_1^e & 0 & N_2^e & 0 & N_3^e \end{bmatrix} \tag{10.94}$$

$$\mathbf{N} = \begin{bmatrix} -\frac{1}{2}\xi\left(1 - \xi\right) & 0 & \frac{1}{2}\xi\left(1 + \xi\right) & 0 & \left(1 + \xi\right)\left(1 - \xi\right) & 0 \\ 0 & -\frac{1}{2}\xi\left(1 - \xi\right) & 0 & \frac{1}{2}\xi\left(1 + \xi\right) & 0 & \left(1 + \xi\right)\left(1 - \xi\right) \end{bmatrix} \tag{10.95}$$

$$\mathbf{N} = \begin{bmatrix} \frac{\xi^2}{2} - \frac{\xi}{2} & 0 & \frac{\xi^2}{2} + \frac{\xi}{2} & 0 & 1 - \xi^2 & 0 \\ 0 & \frac{\xi^2}{2} - \frac{\xi}{2} & 0 & \frac{\xi^2}{2} + \frac{\xi}{2} & 0 & 1 - \xi^2 \end{bmatrix} \tag{10.96}$$

and

$$\begin{bmatrix} u_x^e(\xi) \\ u_y^e(\xi) \end{bmatrix} = \mathbf{N}(\xi) \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{x2}^e \\ u_{y2}^e \\ u_{x3}^e \\ u_{y3}^e \end{bmatrix} \tag{10.97}$$

or if:

$$\mathbf{N} = \begin{bmatrix} N_1^e & N_2^e & N_3^e \end{bmatrix} = \begin{bmatrix} \frac{\xi^2}{2} - \frac{\xi}{2} & \frac{\xi^2}{2} + \frac{\xi}{2} & 1 - \xi^2 \end{bmatrix} \tag{10.98}$$

then:

$$\begin{bmatrix} u_x^e(\xi) \\ u_y^e(\xi) \end{bmatrix} = \mathbf{N}(\xi) \begin{bmatrix} u_{x1}^e & u_{y1}^e \\ u_{x2}^e & u_{y2}^e \\ u_{x3}^e & u_{y3}^e \end{bmatrix} \tag{10.99}$$

shape function derivatives:

$$\mathbf{B} = \frac{\partial \mathbf{N}}{\partial \xi} = \begin{bmatrix} \xi - \frac{1}{2} & 0 & \xi + \frac{1}{2} & 0 & -2\xi & 0 \\ 0 & \xi - \frac{1}{2} & 0 & \xi + \frac{1}{2} & 0 & -2\xi \end{bmatrix} \tag{10.100}$$

the strain-displacement realtion is given by:

$$\varepsilon = \mathbf{Bu} \tag{10.101}$$

Integrating the element at two **Gauss-Legendre** points of the natural coordinate $\xi = [-0.5773502692, 0.5773502692]$ with weights of $w = 1.0$ then numerically:

To get the correct interpolation points and weights, python can be used:

```
1           import numpy as np
2
3           xi, w = np.polynomial.legendre.leggauss(2)
```

for $\xi^1 = -0.5773502692$

$$\mathbf{N}^1 = \begin{bmatrix} 0.45534 & -0.12201 & 0.66667 \end{bmatrix} \tag{10.102}$$

for $\xi^2 = 0.5773502692$

$$\mathbf{N}^2 = \begin{bmatrix} -0.12201 & 0.45534 & 0.66667 \end{bmatrix} \tag{10.103}$$

Shape Functions in global coordinates:

$$\begin{bmatrix} u_{x1}^i \\ u_{y1}^i \end{bmatrix} = \begin{bmatrix} \frac{\xi^2}{2} - \frac{\xi}{2} & 0 & \frac{\xi^2}{2} + \frac{\xi}{2} & 0 & 1 - \xi^2 & 0 \\ 0 & \frac{\xi^2}{2} - \frac{\xi}{2} & 0 & \frac{\xi^2}{2} + \frac{\xi}{2} & 0 & 1 - \xi^2 \end{bmatrix}_{\xi^i} \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{x2}^e \\ u_{y2}^e \\ u_{x3}^e \\ u_{y3}^e \end{bmatrix}$$

$$\tag{10.104}$$

$$\begin{bmatrix} u_x^1 \\ u_y^1 \end{bmatrix} = \begin{bmatrix} 0.45534u_{x1} - 0.12201u_{x2} + 0.66667u_{x3} \\ -0.12201u_{y1} + 0.45534u_{y2} + 0.66667u_{y3} \end{bmatrix} \qquad (10.105)$$

$$\begin{bmatrix} u_x^2 \\ u_y^2 \end{bmatrix} = \begin{bmatrix} -0.12201u_{x1} + 0.45534u_{x2} + 0.66667u_{x3} \\ 0.45534u_{y1} - 0.12201u_{y2} + 0.66667u_{y3} \end{bmatrix} \qquad (10.106)$$

Shape Function Derivatives in Natural Coordinates (there is a row for each integration point if the coordinates are specified as matrix, not a column vector. otherwise for each integration point there is a separate **B** matrix):

$$\frac{\partial \mathbf{N}}{\partial \xi} = \mathbf{B} = \begin{bmatrix} \xi - \frac{1}{2} & 0 & \xi + \frac{1}{2} & 0 & -2\xi & 0 \\ 0 & \xi - \frac{1}{2} & 0 & \xi + \frac{1}{2} & 0 & -2\xi \end{bmatrix} \qquad (10.107)$$

Jacobian:

$$\mathbf{J} = \mathbf{Bu} = \begin{bmatrix} \xi - \frac{1}{2} & 0 & \xi + \frac{1}{2} & 0 & -2\xi & 0 \\ 0 & \xi - \frac{1}{2} & 0 & \xi + \frac{1}{2} & 0 & -2\xi \end{bmatrix} \begin{bmatrix} u_{x1}^e \\ u_{y1}^e \\ u_{x2}^e \\ u_{y2}^e \\ u_{x3}^e \\ u_{y3}^e \end{bmatrix} \qquad (10.108)$$

for $\xi^1 = -0.5773502692$

$$\mathbf{J}^1 = \begin{bmatrix} -1.07735u_{x1} - 0.07735u_{x2} + 1.15470u_3 \\ -1.07735u_{y1} - 0.07735u_{y2} + 1.15470u_{y3} \end{bmatrix} \qquad (10.109)$$

for $\xi^2 = 0.5773502692$

$$\mathbf{J}^2 = \begin{bmatrix} 0.07735u_{x1} + 1.07735u_{x2} - 1.15470u_3 \\ 0.07735u_{y1} + 1.07735u_{y2} - 1.15470u_{y3} \end{bmatrix} \qquad (10.110)$$

Jacobian Determinant (when the **Jacobian matrix** is a vector, its determinant is simply its **length**):

$$\det \mathbf{J} = ||\mathbf{J}|| \qquad (10.111)$$

for $\xi^1 = -0.5773502692$

$$\det \mathbf{J}^1 = \sqrt{(-1.07735u_{x1} - 0.07735u_{x2} + 1.15470u_3)^2 + (-1.07735u_{y1} - 0.07735u_{y2} + 1.15470u_{y3})^2}$$
$$(10.112)$$

for $\xi^2 = 0.5773502692$

$$\det \mathbf{J}^2 = \sqrt{(0.07735u_{x1} + 1.07735u_{x2} - 1.15470u_3)^2 + (0.07735u_{y1} + 1.07735u_{y2} - 1.15470u_{y3})^2}$$
$$(10.113)$$

Jacobian Inverse (when the **Jacobian matrix** is a row vector, its inverse is just its **transpose** divided by its determinant squared):

Moore-Penrose pseudoinverse:

$$\mathbf{J}^{-1} = \mathbf{J}^{+} = \left(\mathbf{J}^T\mathbf{J}\right)^{-1}\mathbf{J}^T \qquad (10.114)$$

for $\xi^1 = -0.5773502692$

$$\mathbf{J}^{1-1} = \frac{1}{\det \mathbf{J}^{1^2}}\mathbf{J}^{1T} \qquad (10.115)$$

for $\xi^2 = 0.5773502692$

$$\mathbf{J}^{2-1} = \frac{1}{\det \mathbf{J}^{2^2}}\mathbf{J}^{2T} \qquad (10.116)$$

Shape function derivatives in global coordinates:

$$\mathbf{B}_g = \mathbf{J}^{-1}\mathbf{B} \qquad (10.117)$$

for $\xi^1 = -0.5773502692$

$$\mathbf{B}_g^1 = \mathbf{J}^{1-1}\mathbf{B}^1 \qquad (10.118)$$

for $\xi^2 = 0.5773502692$

$$\mathbf{B}_g^2 = \mathbf{J}^{2-1}\mathbf{B}^2 \qquad (10.119)$$

Material Stiffness Matrix

$$\mathbf{D} = EA \qquad (10.120)$$

Then the stiffness matrix is composed as:

$$\mathbf{K}^e = \sum_{i=1}^{2} \mathbf{B}_g^{i\,T}\mathbf{D}\mathbf{B_g}^i \det \mathbf{J}^i w^i \qquad (10.121)$$

10.4.6   Python Implementation

```python
#!/usr/bin/python3

import numpy as np
np.set_printoptions(precision=6, suppress=True)

def rod2(xyz, E, A, rho, fx, fy, fz, gauss_points=1):
    print('\nStarting ROD element generation:\n')
    coors = xyz.reshape(-1, 1)
    print('-------------------------------------------------------------
    ↪  Input')
    print('coors =\n{0}'.format(coors))
    print(f'{A = } mm2\n{rho = } t/mm2\n{fx = } N/mm\n{fy = } N/mm' +
          f'\n{fz = } N/mm\n{gauss_points = }\n')

    domain = np.array([[-1],
                       [ 1]], dtype=float)
    print('Domain: {0}\n{1}\n'.format(domain.shape, domain))

    print('----------------------------------------------------
    ↪  Gauss-Legendre')
    # gauss-legendre integration
    ip, iw = np.polynomial.legendre.leggauss(gauss_points)
    print('Gauss-Legendre Integration\nip = {0}\n{1}\n'.format(ip, iw))

    full_domain = np.vstack((domain, ip.reshape(-1, 1)))

    print('-------------------------------------------------- Shape
    ↪  Functions')
    # shape functions in natural coordinates
    xi = ip.T
    print('xi = {0}\n'.format(xi))
    psi = []
    for i in range(xi.shape[0]):
        psi.append(np.array([[1/2 - xi[i]/2, 0, 0, 1/2 + xi[i]/2, 0, 0],
                             [0, 1/2 - xi[i]/2, 0, 0, 1/2 + xi[i]/2, 0],
                             [0, 0, 1/2 - xi[i]/2, 0, 0, 1/2 + xi[i]/2]],
                    dtype=float))
    psi = np.array(psi)
```

```python
36        # psi = psi.transpose(0, 2, 1)
37        print('Shape Functions: {0}\n{1}\n'.format(psi.shape, psi))
38
39        # integration points in global coordinates
40        psi_g = psi @ coors
41        print('Integration Points in '
42              'Global Coordinates: {0}\n{1}\n'.format(psi_g.shape, psi_g))
43
44        # shape function derivatives in natural coordinates
45        dpsi = []
46        for i in range(xi.shape[0]):
47            dpsi.append(np.array([[-1/2, 0, 0, 1/2, 0, 0],
48                                  [0, -1/2, 0, 0, 1/2, 0],
49                                  [0, 0, -1/2, 0, 0, 1/2]], dtype=float))
50        dpsi = np.array(dpsi)
51        print('Shape Functions Derivatives: {0}\n{1}\n'.format(dpsi.shape,
      ↪  dpsi))
52
53        print('-----------------------------------------------------------
      ↪  Jacobians')
54        # Jacobian Matrix
55        jacobi = dpsi @ coors
56        print('Jacobian Matrix: {0}\n{1}\n'.format(jacobi.shape, jacobi))
57
58        # Jacobian Determinants
59        d_jacobi = []
60        for i in range(xi.shape[0]):
61            d_jacobi.append(np.linalg.norm(jacobi[i]))
62        d_jacobi = np.array(d_jacobi)
63        print('Determinant of Jacobian: {0}\n{1}\n'.format(d_jacobi.shape,
      ↪  d_jacobi))
64
65        # Inverse Jacobian
66        i_jacobi = np.linalg.pinv(jacobi)
67        print('Inverse Jacobian Matrix: {0}\n{1}\n'.format(i_jacobi.shape,
      ↪  i_jacobi))
68
69        # Shape Function Derivatives in Global Coordinates
70        dpsi_g = i_jacobi @ dpsi
71        print('Shape Function Derivatives in Global Coordinates:
      ↪  {0}\n{1}\n'.format(dpsi_g.shape, dpsi_g))
```

```
72
73      # material stiffness
74      D = np.array([[E * A]], dtype=float)
75      print('Material Stiffness Matrix: {0}\n{1}\n'.format(D.shape, D))
76
77      # create element stiffness and mass matrix
78      Ke = np.zeros((domain.shape[0] * 3, domain.shape[0] * 3), dtype=float)
79      Me = np.zeros((domain.shape[0] * 3, domain.shape[0] * 3), dtype=float)
80      Fe = np.zeros((domain.shape[0] * 3, 1), dtype=float)
81
82      print('---------------------------------------------------- System
    ↪  Matrices')
83      # iterate over integration points
84      F = np.array([[fx], [fy], [fz]], dtype=float)
85      for i in range(xi.shape[0]):
86          B = dpsi_g[i]
87          N = psi[i]
88
89          Ke += (B.T @ D @ B) * d_jacobi[i] * iw[i]
90          Me += rho * (N.T @ N) * d_jacobi[i] * iw[i]
91          Fe += (N.T @ F) * d_jacobi[i] * iw[i]
92
93      print('Element Stiffness Matrix: {0}\n{1}\n'.format(Ke.shape, Ke))
94      print('Element Mass Matrix: {0}\n{1}\n'.format(Me.shape, Me))
95      print('Element Volume Force Vector: {0}\n{1}\n'.format(Fe.shape, Fe))
96
97      print('----------------------------------------------------
    ↪  Displacements')
98      # xyz displacements at end nodes
99      u = np.array([0.,
100                     0.,
101                     0.,
102                     .001,
103                     .001,
104                     .001], dtype=float).reshape(-1, 1)
105     print('Displacements {0}\n{1}\n'.format(u.shape, u))
106
107     print('----------------------------------------------------------
    ↪  Strains')
108     eps = []
109     for i in range(xi.shape[0]):
```

```
110            eps.append(dpsi_g[i] @ u)
111        eps = np.array(eps)
112        print('Strains {0}\n{1}\n'.format(eps.shape, eps))
113
114        print('------------------------------------------------------------
       ↪  Stresses')
115        sig = D @ eps
116        print('Stresses {0}\n{1}\n'.format(sig.shape, sig))
117
118        return Ke, Me, Fe
119
120
121
122 if __name__ == '__main__':
123     coor = np.array([[    0, 0., 0.],
124                      [1000., 0., 0.]], dtype=float)
125
126     E = 210000.0    # MPa steel
127     A = 200.        # steel
128     rho = 9.81E-9  # steel t/mm3
129
130     fx = 1.   # volumetric continuous load
131     fy = 0.   # volumetric continuous load
132     fz = 1.   # volumetric continuous load
133
134     rod2(coor, E, A, rho, fx, fy, fz, 1)
135
```

10.5   HEX8

10.5.1   Element Formulation

The **Hexahedral** element is a 3-D quadrilateral **trilinear** element, otherwise known as a **brick**. Topologically is hexahedron equivalent to a cube. It has eight corners, twelve edges and six faces. Especially the **HEX8** element has 8 interpolation points.

Natural coordinates

The **natural coordinate system** of a hexahedral element are called **isoparametric hexahedral coordinates**, denoted ξ, η and μ. The coordinates go from -1 to 1 and span from each face to the opposite one. Each coordinate is 0 on the plane at midpoint of opposing faces called the **median** face. This choice of limits is to facilitate the use of standard *Gauss integration formulas*.

Corner Numbering Rules

The eight corners of a hexahedron are locally numbered $1, 2, \ldots, 8$. The corner numbering rule is:

- select one face and numbers the corners of this face $1 \ldots 4$ in a **counterclockwise** direction.

- number the corners directly opposite to corners $1, 2, 3, 4$ as $5, 6, 7, 8$ respectively.

The purpose of this numbering manner is so that a positive volume (or more precisely, a positive **Jacobian determinant** at every point).

The definition of ξ, η and μ can be now made more precise:

- ξ goes from -1 from (center of) face 1485 to $+1$ on face 2376

- η goes from -1 from (center of) face 1265 to $+1$ on face 3487

- μ goes from -1 from (center of) face 1234 to $+1$ on face 5678
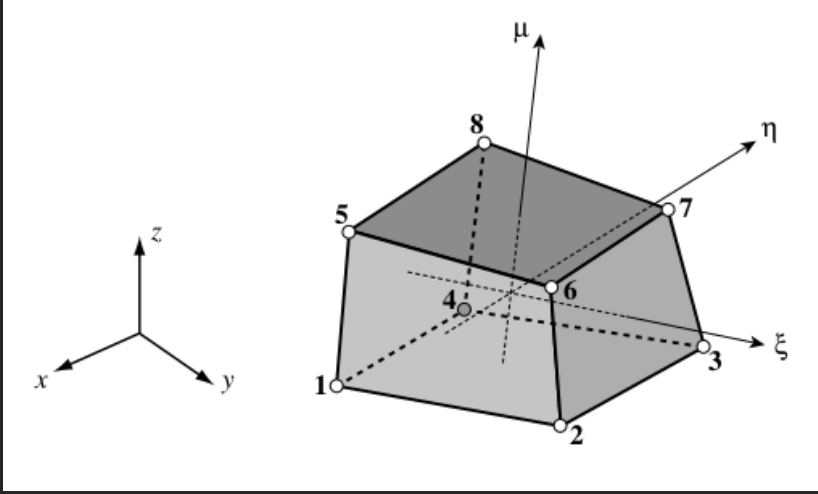
Figure 10.2: The 8-node hexahedron and the natural coordinates η, ξ and μ.

Element Definition

The 8-noded hexahedral element is defined by:

$$
\begin{bmatrix} 1 \\ x \\ y \\ z \\ u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \\ y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 \\ y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 \\ u_{x1} & u_{x2} & u_{x3} & u_{x4} & u_{x5} & u_{x6} & u_{x7} & u_{x8} \\ u_{y1} & u_{y2} & u_{y3} & u_{y4} & u_{y5} & u_{y6} & u_{y7} & u_{y8} \\ u_{z1} & u_{z2} & u_{z3} & u_{z4} & u_{z5} & u_{z6} & u_{z7} & u_{z8} \end{bmatrix} \begin{bmatrix} N_1^e \\ N_2^e \\ N_3^e \\ N_4^e \\ N_5^e \\ N_6^e \\ N_7^e \\ N_8^e \end{bmatrix} \quad (10.122)
$$

The hexahedron corners natural coordinates are:

node	ξ	η	μ
1	-1	-1	-1
2	+1	-1	-1
3	+1	+1	-1
4	-1	+1	-1
5	-1	-1	+1
6	+1	-1	+1
7	+1	+1	+1
8	-1	+1	+1

Table 10.5: Hexahedron corners natural coordinates

The shape functions are:

$$N_1^e = \frac{1}{8}\left(1 - \xi\right)\left(1 - \eta\right)\left(1 - \mu\right)$$

$$N_2^e = \frac{1}{8}\left(1 + \xi\right)\left(1 - \eta\right)\left(1 - \mu\right)$$

$$N_3^e = \frac{1}{8}\left(1 + \xi\right)\left(1 + \eta\right)\left(1 - \mu\right)$$

$$N_4^e = \frac{1}{8}\left(1 - \xi\right)\left(1 + \eta\right)\left(1 - \mu\right)$$

$$N_5^e = \frac{1}{8}\left(1 - \xi\right)\left(1 - \eta\right)\left(1 + \mu\right) \qquad (10.123)$$

$$N_6^e = \frac{1}{8}\left(1 + \xi\right)\left(1 - \eta\right)\left(1 + \mu\right)$$

$$N_7^e = \frac{1}{8}\left(1 + \xi\right)\left(1 + \eta\right)\left(1 + \mu\right)$$

$$N_8^e = \frac{1}{8}\left(1 - \xi\right)\left(1 + \eta\right)\left(1 + \mu\right)$$

The eight formulas can be summarised in a single expression:

$$N_i^e = \frac{1}{8}\left(1 + \xi\xi_i\right)\left(1 + \eta\eta_i\right)\left(1 + \mu\mu_i\right) \qquad (10.124)$$

where ξ_i, η_i and μ_i denote the coordinates of the i-th node.

Partial Derivatives

The calculation of the shape functions derivatives with respect to the natural coordinates:

$$
\begin{aligned}
\frac{\partial N_1^e}{\partial \xi} &= -\frac{1}{8}\left(1-\eta\right)\left(1-\mu\right) \\[2mm]
\frac{\partial N_2^e}{\partial \xi} &= \frac{1}{8}\left(1-\eta\right)\left(1-\mu\right) \\[2mm]
\frac{\partial N_3^e}{\partial \xi} &= \frac{1}{8}\left(1+\eta\right)\left(1-\mu\right) \\[2mm]
\frac{\partial N_4^e}{\partial \xi} &= -\frac{1}{8}\left(1+\eta\right)\left(1-\mu\right) \\[2mm]
\frac{\partial N_5^e}{\partial \xi} &= -\frac{1}{8}\left(1-\eta\right)\left(1+\mu\right) \\[2mm]
\frac{\partial N_6^e}{\partial \xi} &= \frac{1}{8}\left(1-\eta\right)\left(1+\mu\right) \\[2mm]
\frac{\partial N_7^e}{\partial \xi} &= \frac{1}{8}\left(1+\eta\right)\left(1+\mu\right) \\[2mm]
\frac{\partial N_8^e}{\partial \xi} &= -\frac{1}{8}\left(1+\eta\right)\left(1+\mu\right)
\end{aligned}
\tag{10.125}
$$

$$\frac{\partial N_1^e}{\partial \eta} = -\frac{1}{8} \left(1 - \xi\right) \left(1 - \mu\right)$$

$$\frac{\partial N_2^e}{\partial \eta} = -\frac{1}{8} \left(1 + \xi\right) \left(1 - \mu\right)$$

$$\frac{\partial N_3^e}{\partial \eta} = \frac{1}{8} \left(1 + \xi\right) \left(1 - \mu\right)$$

$$\frac{\partial N_4^e}{\partial \eta} = \frac{1}{8} \left(1 - \xi\right) \left(1 - \mu\right)$$

$$\frac{\partial N_5^e}{\partial \eta} = -\frac{1}{8} \left(1 - \xi\right) \left(1 + \mu\right) \tag{10.126}$$

$$\frac{\partial N_6^e}{\partial \eta} = -\frac{1}{8} \left(1 + \xi\right) \left(1 + \mu\right)$$

$$\frac{\partial N_7^e}{\partial \eta} = \frac{1}{8} \left(1 + \xi\right) \left(1 + \mu\right)$$

$$\frac{\partial N_8^e}{\partial \eta} = \frac{1}{8} \left(1 - \xi\right) \left(1 + \mu\right)$$

$$\frac{\partial N_1^e}{\partial \mu} = -\frac{1}{8} \left(1 - \xi\right) \left(1 - \eta\right)$$

$$\frac{\partial N_2^e}{\partial \mu} = -\frac{1}{8} \left(1 + \xi\right) \left(1 - \eta\right)$$

$$\frac{\partial N_3^e}{\partial \mu} = -\frac{1}{8} \left(1 + \xi\right) \left(1 + \eta\right)$$

$$\frac{\partial N_4^e}{\partial \mu} = -\frac{1}{8} \left(1 - \xi\right) \left(1 + \eta\right)$$

$$\frac{\partial N_5^e}{\partial \mu} = \frac{1}{8} \left(1 - \xi\right) \left(1 - \eta\right) \tag{10.127}$$

$$\frac{\partial N_6^e}{\partial \mu} = \frac{1}{8} \left(1 + \xi\right) \left(1 - \eta\right)$$

$$\frac{\partial N_7^e}{\partial \mu} = \frac{1}{8} \left(1 + \xi\right) \left(1 + \eta\right)$$

$$\frac{\partial N_8^e}{\partial \mu} = \frac{1}{8} \left(1 - \xi\right) \left(1 + \eta\right)$$

Note:
The partial derivatives can be also written as:

$$\frac{\partial N_i^e}{\partial \xi} = \frac{1}{8} \frac{\xi_i}{|\xi_i|} (1 + \eta\eta_i)(1 + \mu\mu_i)$$

$$\frac{\partial N_i^e}{\partial \eta} = \frac{1}{8} \frac{\eta_i}{|\eta_i|} (1 + \xi\xi_i)(1 + \mu\mu_i) \tag{10.128}$$

$$\frac{\partial N_i^e}{\partial \mu} = \frac{1}{8} \frac{\mu_i}{|\mu_i|} (1 + \xi\xi_i)(1 + \eta\eta_i)$$

The Jacobian:
The derivatives of the shape functions are given by the usual chain rule formulas:

$$\frac{\partial N_i^e}{\partial x} = \frac{\partial N_i^e}{\partial \xi} \frac{\partial \xi}{\partial x} + \frac{\partial N_i^e}{\partial \eta} \frac{\partial \eta}{\partial x} + \frac{\partial N_i^e}{\partial \mu} \frac{\partial \mu}{\partial x}$$

$$\frac{\partial N_i^e}{\partial y} = \frac{\partial N_i^e}{\partial \xi} \frac{\partial \xi}{\partial y} + \frac{\partial N_i^e}{\partial \eta} \frac{\partial \eta}{\partial y} + \frac{\partial N_i^e}{\partial \mu} \frac{\partial \mu}{\partial y} \tag{10.129}$$

$$\frac{\partial N_i^e}{\partial z} = \frac{\partial N_i^e}{\partial \xi} \frac{\partial \xi}{\partial z} + \frac{\partial N_i^e}{\partial \eta} \frac{\partial \eta}{\partial z} + \frac{\partial N_i^e}{\partial \mu} \frac{\partial \mu}{\partial z}$$

In matrix form:

$$\begin{bmatrix} \frac{\partial N_i^e}{\partial x} \\ \frac{\partial N_i^e}{\partial y} \\ \frac{\partial N_i^e}{\partial z} \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \eta}{\partial x} & \frac{\partial \mu}{\partial x} \\ \frac{\partial \xi}{\partial y} & \frac{\partial \eta}{\partial y} & \frac{\partial \mu}{\partial y} \\ \frac{\partial \xi}{\partial z} & \frac{\partial \eta}{\partial z} & \frac{\partial \mu}{\partial z} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i^e}{\partial \xi} \\ \frac{\partial N_i^e}{\partial \eta} \\ \frac{\partial N_i^e}{\partial \mu} \end{bmatrix} \tag{10.130}$$

The 3×3 matrix above is $\mathbf{J}^{-1}$, the inverse of:

$$\mathbf{J} = \frac{\partial (x, y, z)}{\partial (\xi, \eta, \mu)} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \mu} & \frac{\partial y}{\partial \mu} & \frac{\partial z}{\partial \mu} \end{bmatrix} \tag{10.131}$$

Matrix $\mathbf{J}$ is called the *Jacobian matrix* of (x, y, z) with respect to (ξ, η, μ). In the finite element literature, matrices $\mathbf{J}$ and $\mathbf{J}^{-1}$ are called simply the *Jacobian* and *inverse Jacobian*, respectively, although such a short name us sometimes ambiguous. The notation

$$\mathbf{J} = \frac{\partial (x, y, z)}{\partial (\xi, \eta, \mu)}$$
$$\mathbf{J}^{-1} = \frac{\partial (\xi, \eta, \mu)}{\partial (x, y, z)} \tag{10.132}$$

is standard in multivariable calculus and suggests that the Jacobian may be viewed as a generalisation of the ordinary derivativem to which it reduces for a scalar function $\mathbf{x} = x(\xi)$.

Computing the Jacobian Matrix

The isoparametric definitioin of hexahedron element geometry is:

$$x = x_i N_i^e$$
$$y = y_i N_i^e \tag{10.133}$$
$$z = z_i N_i^e$$

where the summation convention is understood to apply over $i = 1, 2, \ldots, n$, in which n denotes the number of element nodes.

Note: This for a given hexahedral element gives an $n \times 3$ matrix.

Differentiating these relations with respect to the hexahedron coordinates we construct the matrix $\mathbf{J}$ as follows:

$$\mathbf{J} = \begin{bmatrix} x_i \frac{\partial N_i^e}{\partial \xi} & y_i \frac{\partial N_i^e}{\partial \xi} & z_i \frac{\partial N_i^e}{\partial \xi} \\ x_i \frac{\partial N_i^e}{\partial \eta} & y_i \frac{\partial N_i^e}{\partial \eta} & z_i \frac{\partial N_i^e}{\partial \eta} \\ x_i \frac{\partial N_i^e}{\partial \mu} & y_i \frac{\partial N_i^e}{\partial \mu} & z_i \frac{\partial N_i^e}{\partial \mu} \end{bmatrix} \tag{10.134}$$

or:

$$\mathbf{J}_i = \begin{bmatrix} \frac{\partial N_1^e}{\partial \xi} & \frac{\partial N_2^e}{\partial \xi} & \cdots & \frac{\partial N_n^e}{\partial \xi} \\ \frac{\partial N_1^e}{\partial \eta} & \frac{\partial N_2^e}{\partial \eta} & \cdots & \frac{\partial N_n^e}{\partial \eta} \\ \frac{\partial N_1^e}{\partial \mu} & \frac{\partial N_2^e}{\partial \mu} & \cdots & \frac{\partial N_n^e}{\partial \mu} \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_n & y_n & z_n \end{bmatrix} \tag{10.135}$$

Given a point of hexahedron coordinates (ξ, η, μ) the Jacobian $\mathbf{J}$ can be easily formed using the above formula, and numerically inverted to form $\mathbf{J}^{-1}$.

The Strain-Displacement Matrix

Having obtained the shape function derivatives, the matrix $\mathbf{B}$ for hexahedron element displays the usual structure for 3D elements:

$$\mathbf{B} = \mathbf{D}\boldsymbol{\Phi} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} \mathbf{q} & 0 & 0 \\ 0 & \mathbf{q} & 0 \\ 0 & 0 & \mathbf{q} \end{bmatrix} = \begin{bmatrix} \mathbf{q}_x & 0 & 0 \\ 0 & \mathbf{q}_y & 0 \\ 0 & 0 & \mathbf{q}_z \\ \mathbf{q}_y & \mathbf{q}_x & 0 \\ 0 & \mathbf{q}_z & \mathbf{q}_y \\ \mathbf{q}_z & 0 & \mathbf{q}_x \end{bmatrix} \tag{10.136}$$

where:

$$\mathbf{q} = \begin{bmatrix} N_1^e & \cdots & N_n^e \end{bmatrix}$$
$$\mathbf{q}_x = \begin{bmatrix} \frac{\partial N_1^e}{\partial x} & \cdots & \frac{\partial N_n^e}{\partial x} \end{bmatrix}$$
$$\mathbf{q}_y = \begin{bmatrix} \frac{\partial N_1^e}{\partial y} & \cdots & \frac{\partial N_n^e}{\partial y} \end{bmatrix} \tag{10.137}$$
$$\mathbf{q}_z = \begin{bmatrix} \frac{\partial N_1^e}{\partial z} & \cdots & \frac{\partial N_n^e}{\partial z} \end{bmatrix}$$

are row vectors of length n, n being the number of nodes in the element.

Stiffness Matrix Evaluation

The element stiffness Matrix is given by:

$$\mathbf{K}^e = \int_{V^e} \mathbf{B}^T \mathbf{E} \mathbf{B} dV^e \tag{10.138}$$

As in two-dimensional case, this is replaced by a numerical integration formula which now involves a triple loop over conventional Gauss quadrature rules. Assuming that the stress-strain matrix $\mathbf{E}$ is constant over the element,

$$\mathbf{K}^e = \sum_{i=1}^{p_1} \sum_{j=1}^{p_2} \sum_{k=1}^{p_3} w_i w_j w_k \mathbf{B}_{ijk}^T \mathbf{E} \mathbf{B}_{ijk} \mathbf{J}_{ijk} \tag{10.139}$$

Here p_1, p_2 and p_3 are the number of Gauss points in the ξ, η and μ direction, respectively, while $\mathbf{B}_{ijk}$ and $\mathbf{J}_{ijk}$ are abbreviations for:

$$\begin{aligned} \mathbf{B}_{ijk} &= \mathbf{B}\left(\xi_i, \eta_i, \mu_i\right) \\ \mathbf{J}_{ijk} &= det\mathbf{J}\left(\xi_i, \eta_i, \mu_i\right) \end{aligned} \tag{10.140}$$

Usually the number of integration points is taken the same in all directions: $p = p_1 = p_2 = p_3$. The total number of Gauss points is thus p^3. Each point adds at most 6 to the stiffness matrix rank. The minimum rank-sufficient rules for the 8-node and 20-node hexahedra are $p = 2$ and $p = 3$, respectively.

10.5.2   Python implementation

```python
#!/usr/bin/python3

import numpy as np
np.set_printoptions(precision=2) # , suppress=True)

def hex8_stiffness_mass_load(coors: np.ndarray,
                             E: float,
                             nu: float,
                             rho: float,
                             fx: float = 0.,
                             fy: float = 0.,
```

```
12                               fz: float = 0.):
13      """
14      In:
15          coors - 8 x 3 coordinate matrix
16                  [[x1,y1,z1], ... , [xn,yn,zn]
17          E     - Youngs's Modulus
18          nu    - Poisson's Constant
19          rho   - Density
20          fx    - Volumetric Load in x direction
21          fy    - Volumetric Load in y direction
22          fz    - Volumetric Load in z direction
23      """
24      domain = np.array([[-1., -1., -1.],
25                         [1., -1., -1.],
26                         [1., 1., -1.],
27                         [-1., 1., -1.],
28                         [-1., -1., 1.],
29                         [1., -1., 1.],
30                         [1., 1., 1.],
31                         [-1., 1., 1.]], dtype=float)
32      print('Domain: {0}\n{1}'.format(domain.shape, domain))
33
34      # gaussian interpolation
35      gauss_points = 3
36      g_points, g_weights = np.polynomial.legendre.leggauss(gauss_points)
37      integration_points = []
38      interpolation_weights = []
39      for i, xi in enumerate(g_points):
40          for j, eta in enumerate(g_points):
41              for k, mu in enumerate(g_points):
42                  integration_points.append([xi, eta, mu])
43                  interpolation_weights.append(g_weights[i] *
44                                               g_weights[j] *
45                                               g_weights[k])
46      integration_points = np.array(integration_points, dtype=float)
47      interpolation_weights = np.array(interpolation_weights, dtype=float)
48      print('Gauss Integration '
49          '{0}:\n{1}\nWeights:\n{2}'.format(gauss_points,
50                                            integration_points,
51                                            interpolation_weights))
52
```

```
53      full_domain = np.vstack((domain, integration_points))
54
55      # Shape Functions in Natural Coordinates
56      xi = integration_points.T[0]
57      eta = integration_points.T[1]
58      mu = integration_points.T[2]
59      psi = np.zeros((8, integration_points.shape[0]), dtype=float)
60      for i in range(8):
61          psi[i] = 1/8 * (1 + xi * domain[i, 0]) *
62                          (1 + eta * domain[i, 1]) *
63                          (1 + mu * domain[i, 2])
64      psi = psi.T
65      print('Shape Functions: {0}\n{1}'.format(psi.shape, psi))
66
67      # Shape Functions in Global Coordinates
68      psi_g = psi @ coors
69      print('Shape Functions in '
70              'Global Coordinates: {0}\n{1}'.format(psi_g.shape, psi_g))
71
72      # Shape Functions Derivatives in Natural Coordinates
73      dpsi = 1 / 8 * np.array([[(eta - 1.0) * (1.0 - mu),# 1
74                                (xi - 1) * (1 - mu),
75                                -(1 - xi) * (1 - eta)],
76                               [(1 - eta) * (1 - mu),     # 2
77                                (-1 - xi) * (1 - mu),
78                                -(1 + xi) * (1 - eta)],
79                               [(1 + eta) * (1 - mu),     # 3
80                                (1 + xi) * (1 - mu),
81                                -(1 + xi) * (1 + eta)],
82                               [(-1.0 - eta) * (1 - mu), # 4
83                                (1 - xi) * (1 - mu),
84                                -(1 - xi) * (1 + eta)],
85                               [(1 - eta) * (-1 - mu),    # 5
86                                -(1 - xi) * (1 + mu),
87                                (1 - xi) * (1 - eta)],
88                               [(1 - eta) * (1 + mu),     # 6
89                                -(1 + xi) * (1 + mu),
90                                (1 + xi) * (1 - eta)],
91                               [(1 + eta) * (1 + mu),     # 7
92                                (1 + xi) * (1 + mu),
93                                (1 + xi) * (1 + eta)],
```

```
94                                [-(1 + eta) * (1 + mu),    # 8
95                                 (1 - xi) * (1 + mu),
96                                 (1 - xi) * (1 + eta)]])
97     dpsi = dpsi.T
98     print('Shape Functions Derivatives: '
99           '{0}\n{1}'.format(dpsi.shape, dpsi))
100
101    # Jacobian Matrix
102    jacobi = dpsi @ coors
103    print('Jacobian Matrix: {0}\n{1}'.format(jacobi.shape, jacobi))
104
105    # Jacobian Determinants
106    d_jacobi = np.linalg.det(jacobi)
107    print('Determinant of Jacobian: '
108          '{0}\n{1}'.format(d_jacobi.shape, d_jacobi))
109
110    # Inverse Jacobian
111    i_jacobi = np.linalg.inv(jacobi)
112    print('Inverse Jacobian Matrix: '
113          '{0}\n{1}'.format(i_jacobi.shape, i_jacobi))
114
115    # Shape Function Derivatives in Global Coordinates
116    dpsi_g = i_jacobi @ dpsi
117    print('Shape Function Derivatives in Global Coordinates: '
118          '{0}\n{1}'.format(dpsi_g.shape, dpsi_g))
119
120    # Material Stiffness Matrix
121    C = E / ((1.0 + nu) * (1.0 - 2.0 * nu)) *
122        np.array([[1.0 - nu, nu, nu, 0.0, 0.0, 0.0],
123                  [nu, 1.0 - nu, nu, 0.0, 0.0, 0.0],
124                  [nu, nu, 1.0 - nu, 0.0, 0.0, 0.0],
125                  [0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0, 0.0],
126                  [0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0],
127                  [0.0, 0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0]],
128                  dtype=float)
129    print('Material Stiffness Matrix: {0}\n{1}'.format(C.shape, C))
130
131    # Create Element Stiffness and Mass Matrix
132    Ke = np.zeros((domain.size, domain.size), dtype=float)
133    Me = np.zeros((domain.size, domain.size), dtype=float)
134    Fe = np.zeros((domain.size, 1), dtype=float)
```

```
135        # o = np.zeros(domain.shape[0], dtype=float)
136
137        # iterate over Gauss points of domain, * means unpack values
138        for i in range(integration_points.shape[0]):
139            # this is smart but arranges dofs by component, not by node
140            # component-wise dof ordering (x1, .. , y1, .. y4, z1, .. , z4)
141            # B = np.array([
142            #   [*dpsi_g[i, 0, :], *o, *o],
143            #   [*o, *dpsi_g[i, 1, :], *o],
144            #   [*o, *o, *dpsi_g[i, 2, :]],
145            #   [*dpsi_g[i, 2, :], *o, *dpsi_g[i, 0, :]],
146            #   [*o, *dpsi_g[i, 2, :], *dpsi_g[i, 1, :]],
147            #   [*dpsi_g[i, 1, :], *dpsi_g[i, 0, :], *o]], dtype=float)
148
149            # dofs ordered by node
150            # node wise dof ordering (x1, y1, z1, .. , x4, y4, z4)
151            dpx = dpsi_g[i,0,:]
152            dpy = dpsi_g[i,1,:]
153            dpz = dpsi_g[i,2,:]
154            # B = np.array(
155            #
156            ↪  [[dpx[0],0,0,dpx[1],0,0,dpx[2],0,0,dpx[3],0,0,dpx[4],0,0,dpx[5],0,0,dpx[6],0,0,
156            #
156            ↪  [0,dpy[0],0,0,dpy[1],0,0,dpy[2],0,0,dpy[3],0,0,dpy[4],0,0,dpy[5],0,0,dpy[6],0,0
157            #
157            ↪  [0,0,dpz[0],0,0,dpz[1],0,0,dpz[2],0,0,dpz[3],0,0,dpz[4],0,0,dpz[5],0,0,dpz[6],0
158            #
158            ↪  [dpy[0],dpx[0],0,dpy[1],dpx[1],0,dpy[2],dpx[2],0,dpy[3],dpx[3],0,dpy[4],dpx[4],
159            #
159            ↪  [0,dpz[0],dpy[0],0,dpz[1],dpy[1],0,dpz[2],dpy[2],0,dpz[3],dpy[3],0,dpz[4],dpy[4
160            #
160            ↪  [dpz[0],0,dpx[0],dpz[1],0,dpx[1],dpz[2],0,dpx[2],dpz[3],0,dpx[3],dpz[4],0,dpx[4
161            #       dtype=_FLOAT)
162            B = np.array([np.column_stack((dpx,    o,    o)).flatten(),
163                          np.column_stack((  o, dpy,    o)).flatten(),
164                          np.column_stack((  o,    o, dpz)).flatten(),
165                          np.column_stack((dpy, dpx,    o)).flatten(),
166                          np.column_stack((  o, dpz, dpy)).flatten(),
167                          np.column_stack((dpz,    o, dpx)).flatten()],
                          ↪  dtype=float)
168
```

```
169          # this is smart but arranges dofs by component, not by node
170          # component-wise dof ordering (x1, .. , y1, .. y4, z1, .. , z4)
171          # N = np.array([
172          #    [*psi[i], *o, *o],
173          #    [*o, *psi[i], *o],
174          #    [*o, *o, *psi[i]]],
175          #    dtype=float)
176
177          # node wise dof ordering (x1, y1, z1, .. , x4, y4, z4)
178          # N =
    ↪    np.array([[p[0],0,0,p[1],0,0,p[2],0,0,p[3],0,0,p[4],0,0,p[5],0,0,p[6],0,0,p[7],0,0],
179          #
    ↪    [0,p[0],0,0,p[1],0,0,p[2],0,0,p[3],0,0,p[4],0,0,p[5],0,0,p[6],0,0,p[7],0],
180          #
    ↪    [0,0,p[0],0,0,p[1],0,0,p[2],0,0,p[3],0,0,p[4],0,0,p[5],0,0,p[6],0,0,p[7]]],
181          #                 dtype=_FLOAT)
182          N = np.array([np.column_stack((p, o, o)).flatten(),
183                        np.column_stack((o, p, o)).flatten(),
184                        np.column_stack((p, o, o)).flatten()], dtype=float)
185
186          F = np.array([[fx], [fy], [fz]], dtype=float)
187
188          Ke += (B.T @ C @ B) * d_jacobi[i] * interpolation_weights[i]
189          Me += rho * (N.T @ N) * d_jacobi[i] * interpolation_weights[i]
190          Fe += (N.T @ F) * d_jacobi[i] * interpolation_weights[i]
191
192      print('Element Stiffness Matrix: {0}\n{1}'.format(Ke.shape, Ke))
193      print('Element Mass Matrix: {0}\n{1}'.format(Me.shape, Me))
194      print('Element Volume Force Vector: {0}\n{1}'.format(Fe.shape, Fe))
```

And to get stresses:

```
1  #!/usr/bin/python3
2
3  import numpy as np
4  np.set_printoptions(precision=2) # , suppress=True)
5
6  def hex8_stresses(coors: np.ndarray,
7                    disp: np.ndarray,
```

```python
 8                       E: float,
 9                       nu: float):
10        """
11        In:
12            coors - 8 x 3 coordinate matrix
13                    [[x1,y1,z1], ... , [xn,yn,zn]
14            disp  - 8 x 3 coordinate matrix
15                    [[u1,v1,w1], ... , [un,vn,wn]
16            E     - Youngs's Modulus
17            nu    - Poisson's Constant
18        """
19        domain = np.array([[-1., -1., -1.],
20                           [1., -1., -1.],
21                           [1., 1., -1.],
22                           [-1., 1., -1.],
23                           [-1., -1., 1.],
24                           [1., -1., 1.],
25                           [1., 1., 1.],
26                           [-1., 1., 1.]], dtype=float)
27        print('Domain: {0}\n{1}'.format(domain.shape, domain))
28
29        # gaussian interpolation
30        gauss_points = 3
31        g_points, g_weights = np.polynomial.legendre.leggauss(gauss_points)
32        integration_points = []
33        interpolation_weights = []
34        for i, xi in enumerate(g_points):
35            for j, eta in enumerate(g_points):
36                for k, mu in enumerate(g_points):
37                    integration_points.append([xi, eta, mu])
38                    interpolation_weights.append(g_weights[i] *
39                                                 g_weights[j] *
40                                                 g_weights[k])
41        integration_points = np.array(integration_points, dtype=float)
42        interpolation_weights = np.array(interpolation_weights, dtype=float)
43        print('Gauss Integration '
44              '{0}:\n{1}\nWeights:\n{2}'.format(gauss_points,
45                                                integration_points,
46                                                interpolation_weights))
47
48        full_domain = np.vstack((domain, integration_points))
```

```
49
50     # Shape Functions in Natural Coordinates
51     xi = integration_points.T[0]
52     eta = integration_points.T[1]
53     mu = integration_points.T[2]
54     psi = np.zeros((8, integration_points.shape[0]), dtype=float)
55     for i in range(8):
56         psi[i] = 1/8 * (1 + xi * domain[i, 0]) *
57                        (1 + eta * domain[i, 1]) *
58                        (1 + mu * domain[i, 2])
59     psi = psi.T
60     print('Shape Functions: {0}\n{1}'.format(psi.shape, psi))
61
62     # Shape Functions in Global Coordinates
63     psi_g = psi @ coors
64     print('Shape Functions in '
65           'Global Coordinates: {0}\n{1}'.format(psi_g.shape, psi_g))
66
67     # Shape Functions Derivatives in Natural Coordinates
68     dpsi = 1 / 8 * np.array([[(eta - 1.0) * (1.0 - mu),# 1
69                               (xi - 1) * (1 - mu),
70                               -(1 - xi) * (1 - eta)],
71                              [(1 - eta) * (1 - mu),    # 2
72                               (-1 - xi) * (1 - mu),
73                               -(1 + xi) * (1 - eta)],
74                              [(1 + eta) * (1 - mu),    # 3
75                               (1 + xi) * (1 - mu),
76                               -(1 + xi) * (1 + eta)],
77                              [(-1.0 - eta) * (1 - mu), # 4
78                               (1 - xi) * (1 - mu),
79                               -(1 - xi) * (1 + eta)],
80                              [(1 - eta) * (-1 - mu),   # 5
81                               -(1 - xi) * (1 + mu),
82                               (1 - xi) * (1 - eta)],
83                              [(1 - eta) * (1 + mu),    # 6
84                               -(1 + xi) * (1 + mu),
85                               (1 + xi) * (1 - eta)],
86                              [(1 + eta) * (1 + mu),    # 7
87                               (1 + xi) * (1 + mu),
88                               (1 + xi) * (1 + eta)],
89                              [-(1 + eta) * (1 + mu),   # 8
```

```python
90                                (1 - xi) * (1 + mu),
91                                (1 - xi) * (1 + eta)]])
92     dpsi = dpsi.T
93     print('Shape Functions Derivatives: '
94             '{0}\n{1}'.format(dpsi.shape, dpsi))
95
96     # Jacobian Matrix
97     jacobi = dpsi @ coors
98     print('Jacobian Matrix: {0}\n{1}'.format(jacobi.shape, jacobi))
99
100    # Jacobian Determinants
101    d_jacobi = np.linalg.det(jacobi)
102    print('Determinant of Jacobian: '
103            '{0}\n{1}'.format(d_jacobi.shape, d_jacobi))
104
105    # Inverse Jacobian
106    i_jacobi = np.linalg.inv(jacobi)
107    print('Inverse Jacobian Matrix: '
108            '{0}\n{1}'.format(i_jacobi.shape, i_jacobi))
109
110    # Shape Function Derivatives in Global Coordinates
111    dpsi_g = i_jacobi @ dpsi
112    print('Shape Function Derivatives in Global Coordinates: '
113            '{0}\n{1}'.format(dpsi_g.shape, dpsi_g))
114
115    # Material Stiffness Matrix
116    C = E / ((1.0 + nu) * (1.0 - 2.0 * nu)) *
117        np.array([[1.0 - nu, nu, nu, 0.0, 0.0, 0.0],
118                   [nu, 1.0 - nu, nu, 0.0, 0.0, 0.0],
119                   [nu, nu, 1.0 - nu, 0.0, 0.0, 0.0],
120                   [0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0, 0.0],
121                   [0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0],
122                   [0.0, 0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0]],
123                   dtype=float)
124    print('Material Stiffness Matrix: {0}\n{1}'.format(C.shape, C))
125
126    du = disp.T @ np.transpose(dpsi_g, axes=[0, 2, 1])
127
128    exx = du[:, 0, 0]
129    eyy = du[:, 1, 1]
130    ezz = du[:, 2, 2]
```

```python
131     exy = du[:, 0, 1] + du[:, 1, 0]
132     eyz = du[:, 1, 2] + du[:, 2, 1]
133     exz = du[:, 0, 2] + du[:, 2, 0]
134     epsilons = np.array([exx, eyy, ezz, exy, eyz, exz])
135
136     sigmas = (C @ epsilons).T
137     epsilons = epsilons.T
138     print('Element Strains: {0}\n{1}'.format(epsilons.shape, epsilons))
139     print('Element Stresses: {0}\n{1}'.format(stresses.shape, stresses))
```

10.6   TET4 Element

10.6.1   Element Formulation

For a linear tetrahedron no numerical integration is needed (there can be
only one integration point, shape functions derivatives are equal to 1).

The linear tetrahedron is not often used for stress analysis because of its
poor performance. Its main value in structural and solid mechanics is
educational: it serves as a vehicle to introduce the basic steps of formula-
tion of 3D solid elements, particularly with regards to the use of natural
coordinate systems and node numbering conventions. It should be noted
that 3D visualisation is notoriously more difficult than 2D, so it is needed
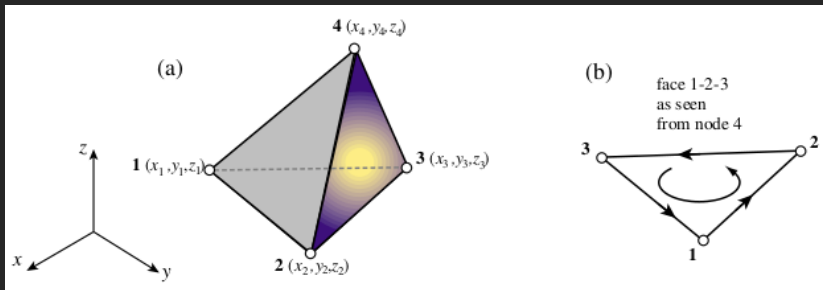to procede more slowly here.

Figure 10.3: (a)The linear tetrahedron element TET4, (b) Node numbering
convention.

Tetrahedron Geometry

The tetrahedron geometry is fully defined by giving the location of the
four corner nodes with respect to Right-Hand Coordinate Sysetm notation
(x, y, z):

10.6.2   Python Implementation

```
1  #!/usr/bin/python3
2
```

```python
 3  import numpy as np
 4  np.set_printoptions(precision=2) # , suppress=True)
 5
 6  def tet4_stiffness_mass_load(coors: np.ndarray,
 7                               E: float,
 8                               nu: float,
 9                               rho: float,
10                               fx: float = 0.,
11                               fy: float = 0.,
12                               fz: float = 0.,
13                               gauss_points = 4):
14      domain = np.array([[1., 0., 0., 0.],
15                         [0., 1., 0., 0.],
16                         [0., 0., 1., 0.],
17                         [0., 0., 0., 1.]], dtype=float)
18      print('Domain: {0}\n{1}'.format(domain.shape, domain))
19
20      # gaussian interpolation
21      integration_points = None
22      integration_weights = None
23      if gauss_points == 1:
24          integration_points = np.array([1/4, 1/4, 1/4, 1/4],
25                                         dtype=float).reshape(1, 4)
26          integration_weights = np.array([1 * 1 * 1 * 1],
27                                          dtype=float) * 1/6
28
29      elif gauss_points == 4:
30          a = 0.58541020
31          b = 0.13819660
32          integration_points = np.array([[a, b, b, b],
33                                          [b, a, b, b],
34                                          [b, b, a, b],
35                                          [b, b, b, a]], dtype=float)
36          integration_weights = np.array([1/4, 1/4, 1/4, 1/4],
37                                          dtype=float) * 1/6
38
39      elif gauss_points == 5:
40          integration_points = np.array([[1/4, 1/4, 1/4, 1/4],
41                                          [1/2, 1/6, 1/6, 1/6],
42                                          [1/6, 1/2, 1/6, 1/6],
43                                          [1/6, 1/6, 1/2, 1/6],
```

```
44                                       [1/6, 1/6, 1/6, 1/2]], dtype=float)
45          integration_weights = np.array([-4/5, 9/20, 9/20, 9/20, 9/20],
46                                  dtype=float) * 1/6
47
48     print('Gauss Integration '
49          '{0}:\n{1}\nWeights:\n{2}'.format(gauss_points,
50                                            integration_points,
51                                            integration_weights))
52
53     full_domain = np.vstack((domain, integration_points))
54
55     # Shape Functions in Natural Coordinates
56     xi = integration_points.T[0]
57     eta = integration_points.T[1]
58     mu = integration_points.T[2]
59     # zeta is not used to enforce
60     # (1 - xi - eta - mu - zeta) = 0
61     # zeta = integration_points.T[3]
62     psi = np.zeros((4, integration_points.shape[0]), dtype=float)
63     # another way of writing that psi = [xi, eta, mu, zeta]
64     # for i in range(4):
65     #     psi[i] = (1 + xi * domain[i, 0]) * (1 + eta * domain[i, 1]) * (1 +
66     ↪   mu * domain[i, 2]) * (1 + zeta * domain[i, 3])
66     psi[0] = xi
67     psi[1] = eta
68     psi[2] = mu
69     psi[3] = 1 - xi - eta - mu  # = zeta
70     psi = psi.T
71     print('Shape Functions: {0}\n{1}'.format(psi.shape, psi))
72
73     # Shape Functions in Global Coordinates
74     psi_g = psi @ coors
75     print('Shape Functions in Global Coordinates: '
76          '{0}\n{1}'.format(psi_g.shape, psi_g))
77
78     # Shape Functions Derivatives in Natural Coordinates
79     dpsi = np.zeros((integration_points.shape[0], 4, 3), dtype=float)
80     dpsi[:,0,0] = 1
81     dpsi[:,1,1] = 1
82     dpsi[:,2,2] = 1
83     dpsi[:,3,:] = -1
```

```python
84      dpsi = dpsi.transpose((0, 2, 1))
85      print('Shape Functions Derivatives: '
86              '{0}\n{1}'.format(dpsi.shape, dpsi))
87
88      # Jacobian Matrix
89      jacobi = dpsi @ coors
90      print('Jacobian Matrix: {0}\n{1}'.format(jacobi.shape, jacobi))
91
92      # Jacobian Determinants
93      d_jacobi = np.linalg.det(jacobi)
94      print('Determinant of Jacobian: '
95              '{0}\n{1}'.format(d_jacobi.shape, d_jacobi))
96
97      # Inverse Jacobian
98      i_jacobi = np.linalg.inv(jacobi)
99      print('Inverse Jacobian Matrix: '
100             '{0}\n{1}'.format(i_jacobi.shape, i_jacobi))
101
102     # Shape Function Derivatives in Global Coordinates
103     dpsi_g = i_jacobi @ dpsi
104     print('Shape Function Derivatives in Global Coordinates: '
105             '{0}\n{1}'.format(dpsi_g.shape, dpsi_g))
106
107     # Material Stiffness Matrix
108     C = E / ((1.0 + nu) * (1.0 - 2.0 * nu)) *
109         np.array([[1.0 - nu, nu, nu, 0.0, 0.0, 0.0],
110                     [nu, 1.0 - nu, nu, 0.0, 0.0, 0.0],
111                     [nu, nu, 1.0 - nu, 0.0, 0.0, 0.0],
112                     [0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0, 0.0],
113                     [0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0],
114                     [0.0, 0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0]],
115                     dtype=float)
116     print('Material Stiffness Matrix: {0}\n{1}'.format(C.shape, C))
117
118     # Create Element Stiffness and Mass Matrix
119     Ke = np.zeros((domain.shape[0] * 3, domain.shape[0] * 3), dtype=float)
120     Me = np.zeros((domain.shape[0] * 3, domain.shape[0] * 3), dtype=float)
121     Fe = np.zeros((domain.shape[0] * 3, 1), dtype=float)
122     o = np.zeros(domain.shape[0], dtype=float)
123
124     # iterate over Gauss points of domain, * means unpack values
```

```
125        for i in range(integration_points.shape[0]):
126            # component-wise dof ordering (x1, .. , y1, .. y4, z1, .. , z4)
127            # B = np.array([[*dpsi_g[i, 0, :], *o, *o],
128            #               [*o, *dpsi_g[i, 1, :], *o],
129            #               [*o, *o, *dpsi_g[i, 2, :]],
130            #               [*dpsi_g[i, 2, :], *o, *dpsi_g[i, 0, :]],
131            #               [*o, *dpsi_g[i, 2, :], *dpsi_g[i, 1, :]],
132            #               [*dpsi_g[i, 1, :], *dpsi_g[i, 0, :], *o]],
133            #               dtype=float)
134
135            # node wise dof ordering (x1, y1, z1, .. , x4, y4, z4)
136            dpx = dpsi_g[i,0,:]
137            dpy = dpsi_g[i,1,:]
138            dpz = dpsi_g[i,2,:]
139            B = np.array(
140                [[dpx[0],0,0,dpx[1],0,0,dpx[2],0,0,dpx[3],0,0],
141                 [0,dpy[0],0,0,dpy[1],0,0,dpy[2],0,0,dpy[3],0],
142                 [0,0,dpz[0],0,0,dpz[1],0,0,dpz[2],0,0,dpz[3]],

143              ↪    [dpy[0],dpx[0],0,dpy[1],dpx[1],0,dpy[2],dpx[2],0,dpy[3],dpx[3],0],

144              ↪    [0,dpz[0],dpy[0],0,dpz[1],dpy[1],0,dpz[2],dpy[2],0,dpz[3],dpy[3]],

145              ↪    [dpz[0],0,dpx[0],dpz[1],0,dpx[1],dpz[2],0,dpx[2],dpz[3],0,dpx[3]],
146                 dtype=float)
147
148            # component-wise dof ordering (x1, .. , y1, .. y4, z1, .. , z4)
149            # N = np.array([[*psi[i], *o, *o],
150            #               [*o, *psi[i], *o],
151            #               [*o, *o, *psi[i]]], dtype=float)
152
153            # node wise dof ordering (x1, y1, z1, .. , x4, y4, z4)
154            N =
               ↪  np.array([[psi[i,0],0,0,psi[i,1],0,0,psi[i,2],0,0,psi[i,3],0,0],

155                          ↪   [0,psi[i,0],0,0,psi[i,1],0,0,psi[i,2],0,0,psi[i,3],0],

156                          ↪   [0,0,psi[i,0],0,0,psi[i,1],0,0,psi[i,2],0,0,psi[i,3]],
157                          dtype=float)
158
159            F = np.array([[fx], [fy], [fz]], dtype=float)
```

```
160
161            print((B.T @ C @ B).shape)
162            print(Ke.shape)
163            Ke += (B.T @ C @ B) * d_jacobi[i] * integration_weights[i]
164            Me += rho * (N.T @ N) * d_jacobi[i] * integration_weights[i]
165            Fe += (N.T @ F) * d_jacobi[i] * integration_weights[i]
166
167        print('Element Stiffness Matrix: {0}\n{1}'.format(Ke.shape, Ke))
168        print('Element Mass Matrix: {0}\n{1}'.format(Me.shape, Me))
169        print('Element Volume Force Vector: {0}\n{1}'.format(Fe.shape, Fe))
```

And to get stresses:

```
1  #!/usr/bin/python3
2
3  import numpy as np
4  np.set_printoptions(precision=2) # , suppress=True)
5
6  def tet4_stresses(coors: np.ndarray,
7                    disp: np.ndarray,
8                    E: float,
9                    nu: float):
10     """
11     In:
12         coors - 4 x 3 coordinate matrix
13               [[x1,y1,z1], ... , [xn,yn,zn]
14         disp  - 4 x 3 coordinate matrix
15               [[u1,v1,w1], ... , [un,vn,wn]
16         E     - Youngs's Modulus
17         nu    - Poisson's Constant
18     """
19     domain = np.array([[1., 0., 0., 0.],
20                        [0., 1., 0., 0.],
21                        [0., 0., 1., 0.],
22                        [0., 0., 0., 1.]], dtype=float)
23     print('Domain: {0}\n{1}'.format(domain.shape, domain))
24
25     # gaussian interpolation
26     integration_points = None
```

```python
27      integration_weights = None
28      if gauss_points == 1:
29          integration_points = np.array([1/4, 1/4, 1/4, 1/4],
30                                         dtype=float).reshape(1, 4)
31          integration_weights = np.array([1 * 1 * 1 * 1],
32                                          dtype=float) * 1/6
33
34      elif gauss_points == 4:
35          a = 0.58541020
36          b = 0.13819660
37          integration_points = np.array([[a, b, b, b],
38                                          [b, a, b, b],
39                                          [b, b, a, b],
40                                          [b, b, b, a]], dtype=float)
41          integration_weights = np.array([1/4, 1/4, 1/4, 1/4],
42                                          dtype=float) * 1/6
43
44      elif gauss_points == 5:
45          integration_points = np.array([[1/4, 1/4, 1/4, 1/4],
46                                          [1/2, 1/6, 1/6, 1/6],
47                                          [1/6, 1/2, 1/6, 1/6],
48                                          [1/6, 1/6, 1/2, 1/6],
49                                          [1/6, 1/6, 1/6, 1/2]], dtype=float)
50          integration_weights = np.array([-4/5, 9/20, 9/20, 9/20, 9/20],
51                                          dtype=float) * 1/6
52
53      print('Gauss Integration '
54            '{0}:\n{1}\nWeights:\n{2}'.format(gauss_points,
55                                              integration_points,
56                                              integration_weights))
57
58      full_domain = np.vstack((domain, integration_points))
59
60      # Shape Functions in Natural Coordinates
61      xi = integration_points.T[0]
62      eta = integration_points.T[1]
63      mu = integration_points.T[2]
64      # zeta is not used to enforce
65      # (1 - xi - eta - mu - zeta) = 0
66      # zeta = integration_points.T[3]
67      psi = np.zeros((4, integration_points.shape[0]), dtype=float)
```

```python
68      # another way of writing that psi = [xi, eta, mu, zeta]
69      # for i in range(4):
70      #     psi[i] = (1 + xi * domain[i, 0]) * (1 + eta * domain[i, 1]) * (1 +
        ↪   mu * domain[i, 2]) * (1 + zeta * domain[i, 3])
71      psi[0] = xi
72      psi[1] = eta
73      psi[2] = mu
74      psi[3] = 1 - xi - eta - mu   # = zeta
75      psi = psi.T
76      print('Shape Functions: {0}\n{1}'.format(psi.shape, psi))
77
78      # Shape Functions in Global Coordinates
79      psi_g = psi @ coors
80      print('Shape Functions in Global Coordinates: '
81            '{0}\n{1}'.format(psi_g.shape, psi_g))
82
83      # Shape Functions Derivatives in Natural Coordinates
84      dpsi = np.zeros((integration_points.shape[0], 4, 3), dtype=float)
85      dpsi[:,0,0] = 1
86      dpsi[:,1,1] = 1
87      dpsi[:,2,2] = 1
88      dpsi[:,3,:] = -1
89      dpsi = dpsi.transpose((0, 2, 1))
90      print('Shape Functions Derivatives: '
91            '{0}\n{1}'.format(dpsi.shape, dpsi))
92
93      # Jacobian Matrix
94      jacobi = dpsi @ coors
95      print('Jacobian Matrix: {0}\n{1}'.format(jacobi.shape, jacobi))
96
97      # Jacobian Determinants
98      d_jacobi = np.linalg.det(jacobi)
99      print('Determinant of Jacobian: '
100           '{0}\n{1}'.format(d_jacobi.shape, d_jacobi))
101
102     # Inverse Jacobian
103     i_jacobi = np.linalg.inv(jacobi)
104     print('Inverse Jacobian Matrix: '
105           '{0}\n{1}'.format(i_jacobi.shape, i_jacobi))
106
107     # Shape Function Derivatives in Global Coordinates
```

```python
108    dpsi_g = i_jacobi @ dpsi
109    print('Shape Function Derivatives in Global Coordinates: '
110          '{0}\n{1}'.format(dpsi_g.shape, dpsi_g))
111
112    # Material Stiffness Matrix
113    C = E / ((1.0 + nu) * (1.0 - 2.0 * nu)) *
114        np.array([[1.0 - nu, nu, nu, 0.0, 0.0, 0.0],
115                  [nu, 1.0 - nu, nu, 0.0, 0.0, 0.0],
116                  [nu, nu, 1.0 - nu, 0.0, 0.0, 0.0],
117                  [0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0, 0.0],
118                  [0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0, 0.0],
119                  [0.0, 0.0, 0.0, 0.0, 0.0, (1.0 - 2.0 * nu) / 2.0]],
120                  dtype=float)
121    print('Material Stiffness Matrix: {0}\n{1}'.format(C.shape, C))
122    du = disp.T @ np.transpose(dpsi_g, axes=[0, 2, 1])
123
124    exx = du[:, 0, 0]
125    eyy = du[:, 1, 1]
126    ezz = du[:, 2, 2]
127    exy = du[:, 0, 1] + du[:, 1, 0]
128    eyz = du[:, 1, 2] + du[:, 2, 1]
129    exz = du[:, 0, 2] + du[:, 2, 0]
130    epsilons = np.array([exx, eyy, ezz, exy, eyz, exz])
131
132    sigmas = (C @ epsilons).T
133    epsilons = epsilons.T
134    print('Element Strains: {0}\n{1}'.format(epsilons.shape, epsilons))
135    print('Element Stresses: {0}\n{1}'.format(stresses.shape, stresses))
```

Chapter 11

Numerics

11.1 Factorisation

11.1.1 Cholesky Factorisation

Cholesky Factorisation is a matrix operation such that a matrix $\mathbf{A}$ is decomposed to a multiplication of a lower triangular martrix $\mathbf{L}$ and its conjugate transpose $\mathbf{L}^T$.

The **Cholesky Factorisation** is applicable to a **Hermitian**, **positive definite** matrices.

$$\mathbf{A} = \mathbf{L}\mathbf{L}^* \tag{11.1}$$

Every Hermitian positive definite matrix (and thus every real-valued symmetric positive definite matrix) has a unique Cholesky factorisation.

If $\mathbf{A}$ can be written as $\mathbf{L}\mathbf{L}^*$ for some invertible $\mathbf{L}$, lower triangular or otherwise, then $\mathbf{A}$ is Hermitian and positive definite.

When $\mathbf{A}$ is a real matrix (hence symmetric and positive definite), the factorisation may be written:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T \tag{11.2}$$

Positive semidefinite matrices

If a Hermitian matrix $\mathbf{A}$ is only positive semidefinite, instead of positive definite, then it still has a decomposition of the form $\mathbf{A} = \mathbf{L}\mathbf{L}^*$ where the diagonal entries of $\mathbf{L}$ are allowed to be zero. The decompostion need not be unique, for example:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{L}\mathbf{L}^* \tag{11.3}$$

$$\mathbf{L} = \begin{bmatrix} 0 & 0 \\ \cos\theta & \sin\theta \end{bmatrix} \tag{11.4}$$

However, if the rank of $\mathbf{A}$ is r, then there is a unique lower triangular $\mathbf{L}$ with exactly r positive diagonal elements adn $n - r$ columns containing all zeros.

Alternatively, the decomposition can be made unique when a pivoting choice is fixed. Formally, if $\mathbf{A}$ is an $n \times n$ positive semidefinite matrix of rank r, then there is at least one permutation matrix $\mathbf{P}$ such that $\mathbf{P}\mathbf{A}\mathbf{P}^T$ has a unique decompostion of the form

$$\mathbf{P}\mathbf{A}\mathbf{P}^T = \mathbf{L}\mathbf{L}^* \tag{11.5}$$

with

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 & \mathbf{0} \\ \mathbf{L}_2 & \mathbf{0} \end{bmatrix} \tag{11.6}$$

where $\mathbf{L}_1$ is a $r \times r$ lower triangular matrix with positive diagonal.

Example

$$\begin{bmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 6 & 1 & 0 \\ -8 & 5 & 3 \end{bmatrix} \begin{bmatrix} 2 & 6 & -8 \\ 0 & 1 & 5 \\ 0 & 0 & 3 \end{bmatrix} \tag{11.7}$$

Applications

Solution of a system of linear equations

The Cholesky decomposition is mainly used for the numerical solution of **linear equations Ax = b**. If **A** is symmetric and positive definite, then we can solve **Ax = b** by first computing the Cholesky decomposition **A = LL***, then solving **Ly = b** for **y** by forward substitution and finally solving **L*x = y** for **x** by back substitution.

$$\begin{aligned}
\mathbf{Ax} &= \mathbf{b} \\
\mathbf{LL^*x} &= \mathbf{b},\ \textit{substitute } \mathbf{L^*x} = \mathbf{y} \\
\mathbf{Ly} &= \mathbf{b},\ \textit{then solve} \\
\mathbf{L^*x} &= \mathbf{y}
\end{aligned} \tag{11.8}$$

For linear systems that can be put into symmetric form, the Cholesky decomposition is the method choice, for superior efficiency and numerical stability. Compared to the **LU** decomposition, it is roughly twice as efficient.

Matrix Inversion

A non-Hermitian matrix **B** can be inverted using the following identity, where **BB*** will always be Hermitian:

$$\mathbf{B}^{-1} = \mathbf{B}^*(\mathbf{BB}^*)^{-1} \tag{11.9}$$

The above stated matrix relation is also called the **pseudoinverse** of a matrix. The following equality holds:

$$\mathbf{B}^*(\mathbf{BB}^*)^{-1} = (\mathbf{B}^*\mathbf{B})^{-1}\mathbf{B}^* \tag{11.10}$$

The best way to compute the is to use . With **A** being $\mathbf{A} = \mathbf{USV}^*$, where **U** and **V** (both $n \times n$) are orthogonal and **S** ($m \times n$) is with real, non-negative *singular values* $\sigma_{i,1} = 1, \ldots, n$. We find that

$$\mathbf{A}^* = \mathbf{V}(\mathbf{S}^*\mathbf{S})^{-1}\mathbf{S}^*\mathbf{U}^* \tag{11.11}$$

If the rank r of **A** is smaller than n, the inverse of $\mathbf{S}^*\mathbf{S}$ does not exist, and one uses only the first r singular values. **S** then becomes an $r \times r$ matrix and **U**, **V** shrink accordingly.

11.2   Linear Regression

11.2.1   Derivation

Linear Regression is an approach for predicting a response using a **single feature**. In linear regression we assume that two variables i. e. independent and dependent are linearly related.

Generalrly we define **x** as a **feature vector**:

$$\mathbf{x} = [x_1, x_2, ..., x_n] \tag{11.12}$$

and **y** as **response vector**:

$$\mathbf{y} = [y_1, y_2, ..., y_n] \tag{11.13}$$

then a predicted response:

$$h(x_i) = \beta_0 + \beta_1 \times x_i \tag{11.14}$$

where:
x_i represents the i-th observation
β_1 represents the slope, and
β_0 represents the intercept of the regression line.

To predict a new value y_i, we need to consider:

$$y_i = \beta_0 + \beta_1 \times x_i + \epsilon_i = h(x_i) + \epsilon_i \tag{11.15}$$

where:
ϵ_i is the **error** of the estimate.

The error function is therefore obtained:

$$\epsilon_i = y_i - h(x_i) \tag{11.16}$$

The β_0 and β_1 coefficients are obtained by minimizing the residual **error** of the estimate using e.g. the **least square method**:

$$J(\beta_0, \beta_1) = \frac{1}{2n} \sum_{i=1}^{n} \epsilon_i^2$$

$$= \frac{1}{2n} \sum_{i=1}^{n} (y_i - h(x_i))^2 \tag{11.17}$$

$$= \frac{1}{2n} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 \times x_i)^2$$

To find the coefficients we can derviate the cost function, first with regards to β_0:

$$\frac{\partial J}{\partial \beta_0} \left[\frac{1}{2n} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 \times x_i)^2 \right] = 0 \tag{11.18}$$

where using the chain rule we can write:

$$u = y_i - \beta_0 - \beta_1 \times x_i$$

$$J = \frac{1}{2n} \sum_{i=1}^{n} u^2$$

$$\frac{\partial J}{\partial \beta_0} = \frac{\partial J}{\partial u} \times \frac{\partial u}{\partial \beta_0}$$

$$\frac{\partial J}{\partial u} = 2u$$

$$\frac{\partial u}{\partial \beta_0} = -1$$

$$\frac{\partial J}{\partial \beta_0} = \frac{1}{2n} \times (-2) \times \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 \times x_i) \qquad (11.19)$$

$$0 = \frac{1}{n} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 \times x_i)$$

$$0 = \frac{1}{n} \sum_{i=1}^{n} y_i - \frac{1}{n} \sum_{i=1}^{n} \beta_0 - \frac{1}{n} \beta_1 \sum_{i=1}^{n} x_i$$

$$0 = \bar{y}_i - \beta_0 - \beta_1 \bar{x}_i$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

To find β_1:

$$\frac{\partial J}{\partial \beta_1} \left[\frac{1}{2n} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 \times x_i)^2 \right] = 0 \qquad (11.20)$$

$$u = y_i - \beta_0 - \beta_1 \times x_i$$

$$J = \frac{1}{2n} \sum_{i=1}^{n} u^2$$

$$\frac{\partial J}{\partial \beta_1} = \frac{\partial J}{\partial u} \times \frac{\partial u}{\partial \beta_1}$$

$$\frac{\partial J}{\partial u} = 2u$$

$$\frac{\partial u}{\partial \beta_1} = -x \qquad (11.21)$$

$$\frac{\partial J}{\partial \beta_1} = \frac{1}{2n} \times (-2) \sum_{i=1}^{n} x_i \times (y_i - \beta_0 - \beta_1 x_i)$$

$$0 = \frac{1}{n} \sum_{i=1}^{n} x_i (y_i - \beta_0 - \beta_1 x_i)$$

$$0 = \frac{1}{n} \sum_{i=1}^{n} \left(x_i y_i - \beta_0 x_i - \beta_1 x_i^2 \right)$$

Substituting the value of β_0:

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

$$0 = \frac{1}{n} \sum_{i=1}^{n} \left(x_i y_i - (\bar{y} - \beta_1 \bar{x}) x_i - \beta_1 x_i^2 \right)$$

$$0 = \frac{1}{n} \sum_{i=1}^{n} \left(x_i y_i - \bar{y} x_i + \beta_1 \bar{x} x_i - \beta_1 x_i^2 \right) \qquad (11.22)$$

$$0 = \frac{1}{n} \left(\sum_{i=1}^{n} (x_i y_i - \bar{y} x_i) - \beta_1 \sum_{i=1}^{n} \left(x_i^2 - \bar{x} x_i \right) \right)$$

$$\beta_1 = \frac{\frac{1}{n} \sum_{i=1}^{n} (x_i y_i - \bar{y} x_i)}{\frac{1}{n} \sum_{i=1}^{n} \left(x_i^2 - \bar{x} x_i \right)}$$

Simplifying the formula using:

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$$

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$(11.23)$$

we get:

$$\beta_1 = \frac{\frac{1}{n} \sum_{i=1}^{n} x_i y_i - \bar{x}\bar{y}}{\frac{1}{n} \sum_{i=1}^{n} x_i^2 - \bar{x}\bar{x}}$$

$$\beta_1 = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}$$

$$(11.24)$$

11.2.2   Result

So, to recapitulate, to get a linear regression coefficients for a dataset:

$$\mathbf{x} = [x_1, x_2, ..., x_n]$$

$$\mathbf{y} = [y_1, y_2, ..., y_n]$$

$$(11.25)$$

we may use equations:

$$\beta_1 = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

$$(11.26)$$

with an error:

$$\epsilon_i = y_i - \beta_0 - \beta_1 x_i$$

$$(11.27)$$

Index