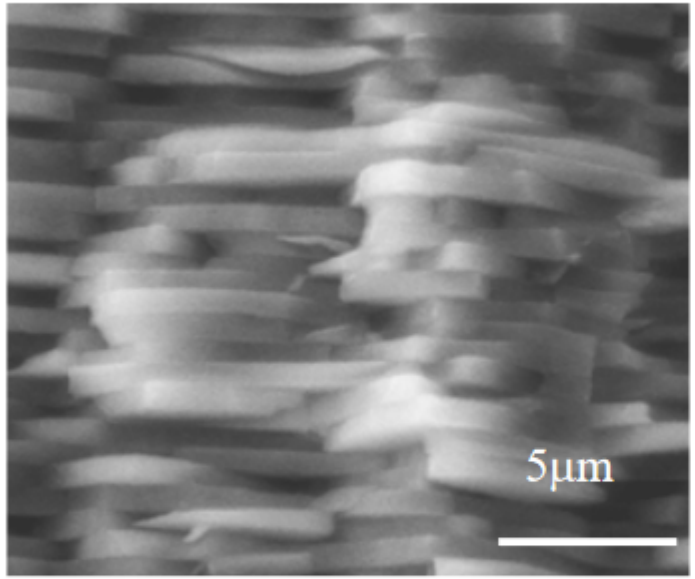
Introduction to Finite Element Method and FEniCS

Panchatcharam Mariappan
Assistant Professor

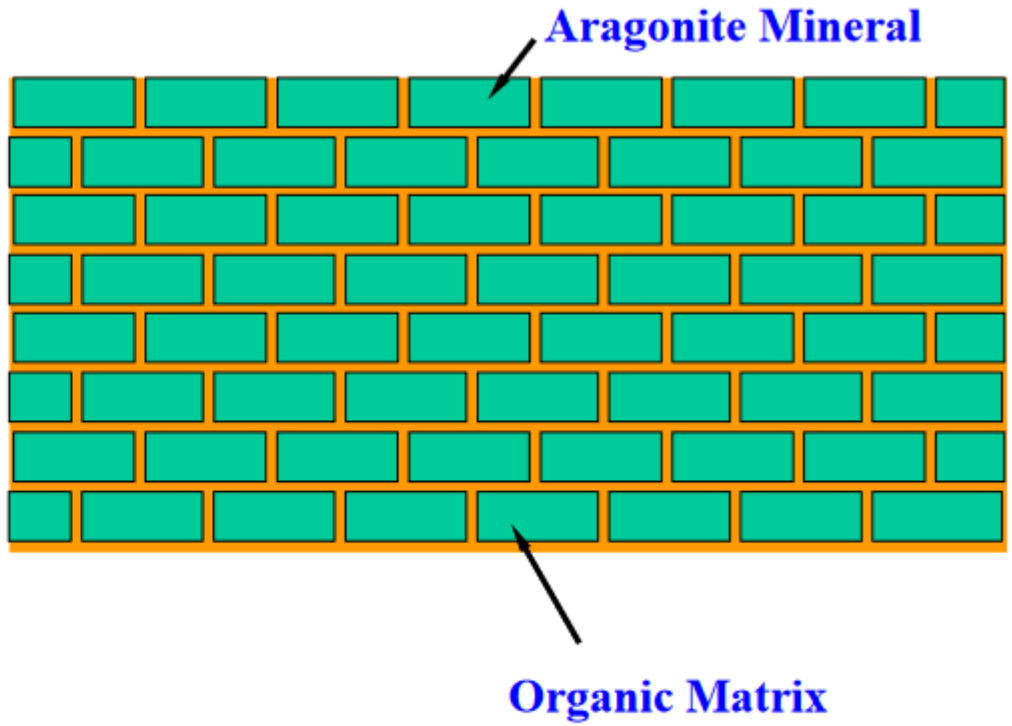
IIT Tirupati

Finite Element Method: 2D Plane Problems

2D Nacre Structure



3D nacre structure



2D simplification

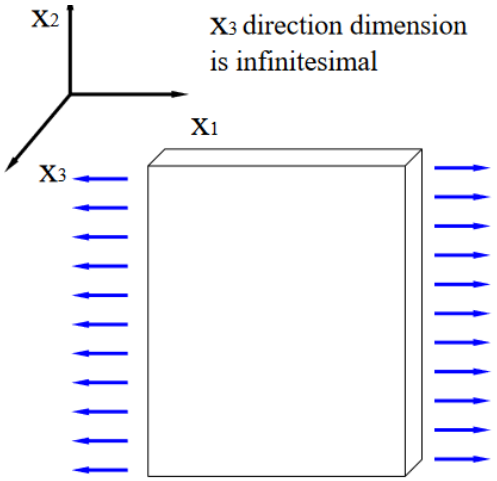
Stress and Strain

Plane Stress and Plane Strain:

Plane stress: All the stress components associate with 3-direction are zero.

$$\sigma_{33} = \sigma_{13} = \sigma_{23} = 0 \quad \sigma_{11} \neq 0 \quad \sigma_{22} \neq 0 \quad \sigma_{12} \neq 0$$

$$F_3 = 0$$

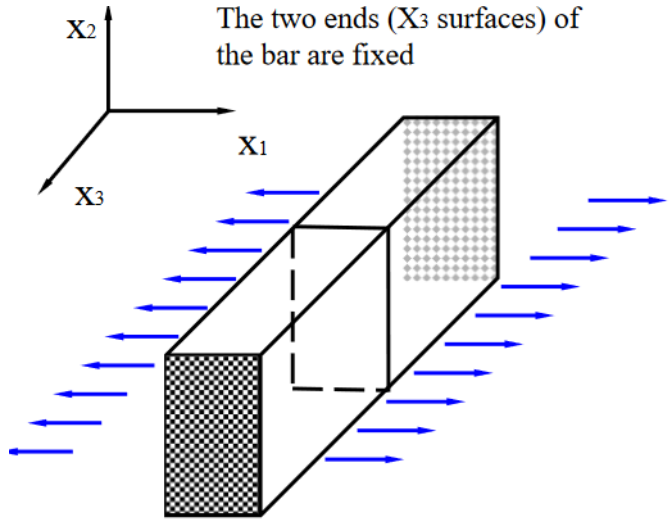


Plane Stress

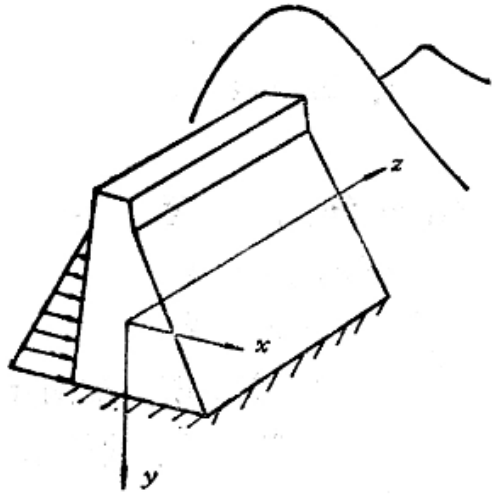
Plane strain: All the strain components associate with 3-direction are zero.

$$\varepsilon_{33} = \varepsilon_{13} = \varepsilon_{23} = 0 \quad \varepsilon_{11} \neq 0 \quad \varepsilon_{22} \neq 0 \quad \varepsilon_{12} \neq 0$$

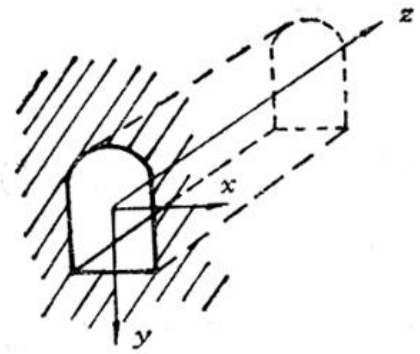
$$u_3 = 0$$



Plane Strain



Dam



Tunnel

Strain-Stress Relationship for Plane Problems

Plane Stress Problem: $\sigma_{33} = \sigma_{13} = \sigma_{23} = 0 \quad \sigma_{11} \neq 0 \quad \sigma_{22} \neq 0 \quad \sigma_{12} \neq 0$

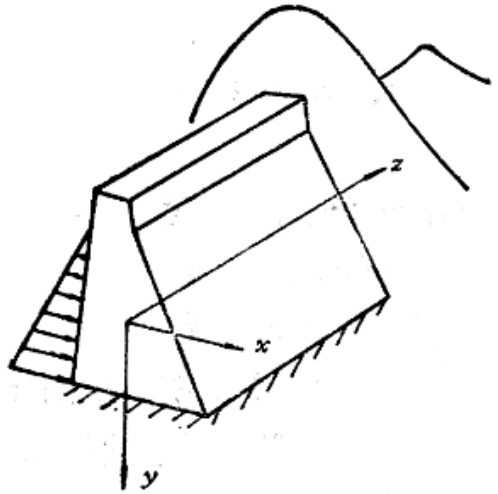
$$\varepsilon_{11} = \frac{1}{E}\left[\sigma_{11} - \nu\sigma_{22}\right] \qquad \varepsilon_{22} = \frac{1}{E}\left[\sigma_{22} - \nu\sigma_{11}\right]$$

$$\varepsilon_{33} = \frac{-\nu}{E}\left(\sigma_{11} + \sigma_{22}\right) \qquad \varepsilon_{12} = \frac{1-\nu}{E}\sigma_{12}$$

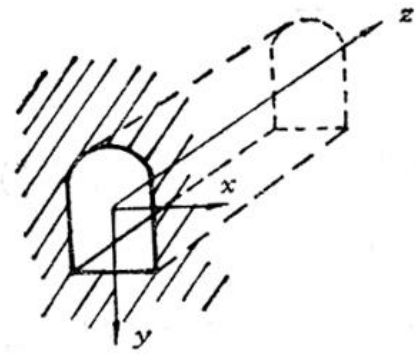
$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{Bmatrix} = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ & 1 & 0 \\ & & 1-\nu \end{bmatrix}\begin{Bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{12} \end{Bmatrix}$$

$$\{\sigma\} = [D]\{\varepsilon\}$$

Stress and Strain



Dam



Tunnel

Plane Strain Problem: $\varepsilon_{33} = \varepsilon_{13} = \varepsilon_{23} = 0 \quad \varepsilon_{11} \neq 0 \quad \varepsilon_{22} \neq 0 \quad \varepsilon_{12} \neq 0$

$$\varepsilon_{11} = \frac{1}{E}\left[\sigma_{11} - \nu(\sigma_{22} + \sigma_{33})\right] \qquad \varepsilon_{22} = \frac{1}{E}\left[\sigma_{22} - \nu(\sigma_{11} + \sigma_{33})\right]$$

$$0 = \frac{1}{E}\left[\sigma_{33} - \nu(\sigma_{11} + \sigma_{22})\right] \qquad \varepsilon_{12} = \frac{1-\nu}{E}\sigma_{12}$$

$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{Bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ & 1-\nu & 0 \\ & & 1-2\nu \end{bmatrix} \begin{Bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{12} \end{Bmatrix}$$

$$\{\sigma\} = [D]\{\varepsilon\}$$

Stress and Strain

$$\{\sigma\} = [D]\{\varepsilon\}$$

Plane Stress

$$[D] = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ & 1 & 0 \\ & & 1-\nu \end{bmatrix}$$

Plane Strain

$$[D] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & \frac{\nu}{(1-\nu)} & 0 \\ & 1 & 0 \\ & & \frac{1-2\nu}{(1-\nu)} \end{bmatrix}$$

Linear Elements 3 nodes

Quadratic Elements 6 nodes

Rectangular Elements

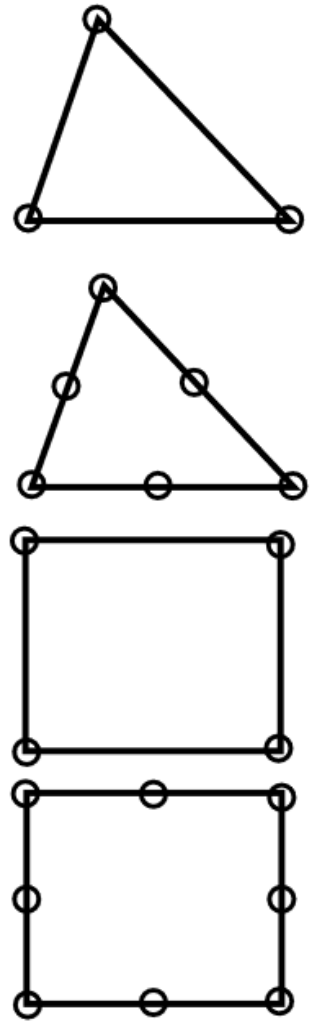
Linear Elements 4 nodes

Quadratic Elements 8 nodes

Element Types

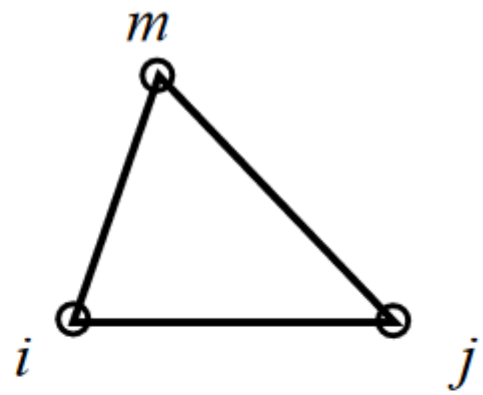
Triangular Elements
- Linear Elements
- Quadratic Elements

Rectangular Elements
- Linear Elements
- Quadratic Elements



Shape Functions

Linear Triangular Element



1. Interpolation of displacement

$$u = a_1 + a_2 x + a_3 y$$

$$v = a_4 + a_5 x + a_6 y$$

$$\begin{Bmatrix} u \\ v \end{Bmatrix} = \begin{bmatrix} 1 & x & y & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & x & y \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{Bmatrix}$$

Shape Functions

$$u_i = a_1 + a_2 x_i + a_3 y_i$$
$$u_j = a_1 + a_2 x_j + a_3 y_j$$
$$u_m = a_1 + a_2 x_m + a_3 y_m$$

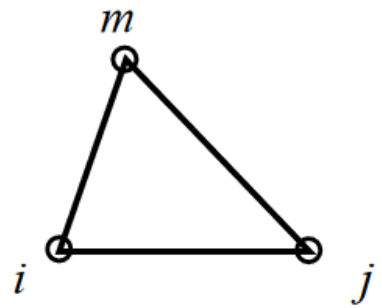
$$\Rightarrow \quad \begin{Bmatrix} u_i \\ u_j \\ u_m \end{Bmatrix} = \begin{bmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_m & y_m \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix}$$

$$v_i = a_4 + a_5 x_i + a_6 y_i$$
$$v_j = a_4 + a_5 x_j + a_6 y_j$$
$$v_m = a_4 + a_5 x_m + a_6 y_m$$

Shape Functions

Linear Triangular Element

1. Interpolation of displacement



In order to get a_1, a_2, a_3, we need to solve the equations:

$$\begin{Bmatrix} u_i \\ u_j \\ u_m \end{Bmatrix} = \begin{bmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_m & y_m \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} \rightarrow \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{bmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_m & y_m \end{bmatrix}^{-1} \begin{Bmatrix} u_i \\ u_j \\ u_m \end{Bmatrix}$$

$$2A = \begin{vmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_m & y_m \end{vmatrix}$$

$$\alpha_i = x_j y_m - x_m y_j$$

$$\beta_i = y_j - y_m$$

$$\gamma_i = x_m - x_j$$

$$\rightarrow \quad \begin{bmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_m & y_m \end{bmatrix}^{-1} = \frac{1}{2A} \begin{bmatrix} \alpha_i & \alpha_j & \alpha_m \\ \beta_i & \beta_j & \beta_m \\ \gamma_i & \gamma_j & \gamma_m \end{bmatrix}$$

Shape Functions

$$\begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{bmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_m & y_m \end{bmatrix}^{-1} \begin{Bmatrix} u_i \\ u_j \\ u_m \end{Bmatrix} \Rightarrow \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \frac{1}{2A} \begin{bmatrix} \alpha_i & \alpha_j & \alpha_m \\ \beta_i & \beta_j & \beta_m \\ \gamma_i & \gamma_j & \gamma_m \end{bmatrix} \begin{Bmatrix} u_i \\ u_j \\ u_m \end{Bmatrix}$$

$$\begin{Bmatrix} a_4 \\ a_5 \\ a_6 \end{Bmatrix} = \frac{1}{2A} \begin{bmatrix} \alpha_i & \alpha_j & \alpha_m \\ \beta_i & \beta_j & \beta_m \\ \gamma_i & \gamma_j & \gamma_m \end{bmatrix} \begin{Bmatrix} v_i \\ v_j \\ v_m \end{Bmatrix}$$

$$u(x,y) = a_1 + a_2 x + a_3 y$$

$$v(x,y) = a_4 + a_5 x + a_6 y$$

$$u(x,y) = \frac{1}{2A}[\underbrace{\alpha_i u_i + \alpha_j u_j + \alpha_m u_m}_{a_1} + \underbrace{(\beta_i u_i + \beta_j u_j + \beta_m u_m)}_{a_2} x + \underbrace{(\gamma_i u_i + \gamma_j u_j + \gamma_m u_m)}_{a_3} y]$$

Shape Functions

$$u(x,y)=\frac{1}{2A}\left[(\alpha_i+\beta_i x+\gamma_i y)u_i+(\alpha_j+\beta_j x+\gamma_j y)u_j+(\alpha_m+\beta_m x+\gamma_m y)u_m\right]$$

$$v(x,y)=\frac{1}{2A}\left[(\alpha_i+\beta_i x+\gamma_i y)v_i+(\alpha_j+\beta_j x+\gamma_j y)v_j+(\alpha_m+\beta_m x+\gamma_m y)v_m\right]$$

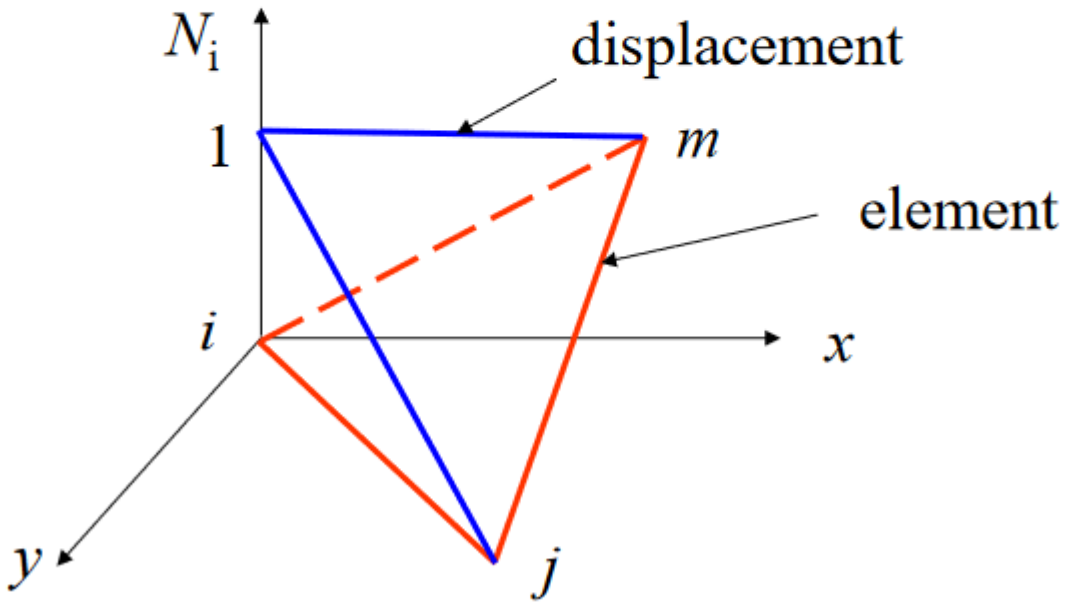
$$u(x,y)=N_i u_i+N_j u_j+N_m u_m$$

$$v(x,y)=N_i v_i+N_j v_j+N_m v_m$$

Shape Functions

Properties of N_i

$$N_i = \begin{cases} 1 & at \quad (x,y) = (x_i, y_i) \\ 0 & at \quad (x,y) = (x_j, y_j) \; or \; (x_m, y_m) \end{cases}$$



Shape Functions

$$u(x,y) = N_i u_i + N_j u_j + N_m u_m$$

$$v(x,y) = N_i v_i + N_j v_j + N_m v_m$$

$$\{\varepsilon\} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} = \begin{Bmatrix} \dfrac{\partial u}{\partial x} \\ \dfrac{\partial v}{\partial y} \\ \left(\dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x} \right) \end{Bmatrix}$$

$$\frac{\partial u}{\partial x} = \frac{\partial N_i}{\partial x} u_i + \frac{\partial N_j}{\partial x} u_j + \frac{\partial N_m}{\partial x} u_m$$

$$\frac{\partial v}{\partial y} = \frac{\partial N_i}{\partial y} v_i + \frac{\partial N_j}{\partial y} v_j + \frac{\partial N_m}{\partial y} v_m$$

$$\frac{\partial u}{\partial y} =$$

$$\frac{\partial v}{\partial x} =$$

Shape Functions

$$N_i = \frac{1}{2A}(\alpha_i + \beta_i x + \gamma_i y) \qquad N_j = \frac{1}{2A}(\alpha_j + \beta_j x + \gamma_j y) \qquad N_m = \frac{1}{2A}(\alpha_m + \beta_m x + \gamma_m y)$$

$$\frac{\partial N_i}{\partial x} = \frac{\beta_i}{2A} \qquad \frac{\partial N_i}{\partial y} = \frac{\gamma_i}{2A}$$

$$\frac{\partial N_j}{\partial x} = \frac{\beta_j}{2A} \qquad \frac{\partial N_j}{\partial y} = \frac{\gamma_j}{2A}$$

$$\frac{\partial N_m}{\partial x} = \frac{\beta_m}{2A} \qquad \frac{\partial N_m}{\partial y} = \frac{\gamma_m}{2A}$$

$$\frac{\partial u}{\partial x} = \frac{1}{2A}\left(\beta_i u_i + \beta_j u_j + \beta_m u_m\right)$$

$$\frac{\partial v}{\partial y} = \frac{1}{2A}\left(\gamma_i v_i + \gamma_j v_j + \gamma_m v_m\right)$$

$$\frac{\partial u}{\partial y} = \frac{1}{2A}\left(\gamma_i u_i + \gamma_j u_j + \gamma_m u_m\right)$$

$$\frac{\partial v}{\partial x} = \frac{1}{2A}\left(\beta_i v_i + \beta_j v_j + \beta_m v_m\right)$$

Shape Functions

$$\{\varepsilon\} = \begin{Bmatrix} \dfrac{\partial u}{\partial x} \\ \dfrac{\partial v}{\partial y} \\ \left(\dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x}\right) \end{Bmatrix}$$

$$\{\varepsilon\} = \frac{1}{2A} \begin{Bmatrix} \beta_i u_i + \beta_j u_j + \beta_m u_m \\ \gamma_i v_i + \gamma_j v_j + \gamma_m v_m \\ \gamma_i u_i + \gamma_j u_j + \gamma_m u_m + \beta_i v_i + \beta_j v_j + \beta_m v_m \end{Bmatrix}$$

Shape Functions

$$\{\varepsilon\} = \frac{1}{2A}\begin{Bmatrix} \beta_i u_i + \beta_j u_j + \beta_m u_m \\ \gamma_i v_i + \gamma_j v_j + \gamma_m v_m \\ \gamma_i u_i + \gamma_j u_j + \gamma_m u_m + \beta_i v_i + \beta_j v_j + \beta_m v_m \end{Bmatrix}$$

$$\{\varepsilon\} = \frac{1}{2A}\begin{bmatrix} \beta_i & 0 & \beta_j & 0 & \beta_m & 0 \\ 0 & \gamma_i & 0 & \gamma_j & 0 & \gamma_m \\ \gamma_i & \beta_i & \gamma_j & \beta_j & \gamma_m & \beta_m \end{bmatrix}\begin{Bmatrix} u_i \\ v_i \\ u_j \\ v_j \\ u_m \\ v_m \end{Bmatrix}$$

Shape Functions

$$\{\varepsilon\} = \frac{1}{2A}\begin{bmatrix} \beta_i & 0 & \beta_j & 0 & \beta_m & 0 \\ 0 & \gamma_i & 0 & \gamma_j & 0 & \gamma_m \\ \gamma_i & \beta_i & \gamma_j & \beta_j & \gamma_m & \beta_m \end{bmatrix} \begin{Bmatrix} u_i \\ v_i \\ u_j \\ v_j \\ u_m \\ v_m \end{Bmatrix}$$

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = [D] \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix}$$

$$\{\varepsilon\} = [B]\{d\}$$

Minimum Potential Energy

Principle of minimum potential energy

$$\delta \Pi_p = 0$$

$$\Pi_p = \int_B \left[U(u_i) + \Phi(u_i) \right] dV + \int_{\partial B} \Psi(u_i) dS$$

$$U(u_i) = \frac{1}{2} \sigma_{ij} \varepsilon_{ij}$$

$$\rightarrow \quad U = \frac{1}{2} \{\varepsilon\}^T \{\sigma\} = \frac{1}{2} \{\varepsilon\}^T [D] \{\varepsilon\} = \frac{1}{2} ([B]\{d\})^T [D][B]\{d\}$$

Shape Functions

$$\Phi(u,v) = -f_1 u - f_2 v = -\{w\}^T \{f\}$$

$$\Psi(u,v) = -\overline{T}_1 u - \overline{T}_2 v = -\{w\}^T \{\overline{T}\}$$

$$\{w\} = \begin{Bmatrix} u \\ v \end{Bmatrix}$$

$$u = N_i u_i + N_j u_j + N_m u_m$$

$$v = N_i v_i + N_j v_j + N_m v_m$$

Shape Functions

$$\Pi_p^{\,e} = \int_{B^e} \left[U(u_i) + \Phi(u_i) \right] dV + \int_{\partial B^e} \Psi(u_i) dS$$

$$\Pi_p^{\,e} = \frac{1}{2} \int_{B^e} \{d\}^T [B]^T [D][B] \{d\} dV$$

$$- \int_{B^e} \{d\}^T [N]^T \{f\} dV - \int_{\partial B^e} \{d\}^T [N]^T \{\overline{T}\} dS$$

Shape Functions

$$\delta \Pi_p = 0 \quad \Longleftrightarrow \quad \frac{\partial \Pi_p}{\partial \{d\}} = 0$$

$$\Pi_p{}^e = \frac{1}{2}\int_{B^e} \{d\}^T [B]^T [D][B]\{d\} dV$$

$$- \int_{B^e} \{d\}^T [N]^T \{f\} dV - \int_{\partial B^e} \{d\}^T [N]^T \{\overline{T}\} dS$$

Stiffness Matrix

Since $\{d\}$ is the nodal displacement vector, it is a constant vector

$$\int_{B^e} [B]^T [D][B] dV \{d\} - \int_{B^e} [N]^T \{f\} dV - \int_{\partial B^e} [N]^T \{\overline{T}\} dS = 0$$

$$\beta_i = y_j - y_m$$

$$\gamma_i = x_m - x_j$$

$$\{\varepsilon\} = \frac{1}{2A} \underbrace{\begin{bmatrix} \beta_i & 0 \\ 0 & \gamma_i \\ \gamma_i & \beta_i \end{bmatrix}}_{[B_i]} \underbrace{\begin{bmatrix} \beta_j & 0 \\ 0 & \gamma_j \\ \gamma_j & \beta_j \end{bmatrix}}_{[B_j]} \underbrace{\begin{bmatrix} \beta_m & 0 \\ 0 & \gamma_m \\ \gamma_m & \beta_m \end{bmatrix}}_{[B_m]} \begin{Bmatrix} u_i \\ v_i \\ u_j \\ v_j \\ u_m \\ v_m \end{Bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} d_i \\ \\ d_j \\ \\ d_m \\ \end{matrix}$$

Stiffness Matrix

$$[B_i] = \frac{1}{2A}\begin{bmatrix} \beta_i & 0 \\ 0 & \gamma_i \\ \gamma_i & \beta_i \end{bmatrix}$$

$$\beta_i = y_j - y_m$$

$$\gamma_i = x_m - x_j$$

$$[k]^e = tA\begin{bmatrix}[B_i]^T & [B_j]^T & [B_m]^T\end{bmatrix}[D]\begin{bmatrix}[B_i] \\ [B_j] \\ [B_m]\end{bmatrix}$$

$$= tA\begin{bmatrix} [B_i]^T[D][B_i] & [B_i]^T[D][B_j] & [B_i]^T[D][B_m] \\ [B_j]^T[D][B_i] & [B_j]^T[D][B_j] & [B_j]^T[D][B_m] \\ [B_i]^T[D][B_m] & [B_j]^T[D][B_m] & [B_m]^T[D][B_m] \end{bmatrix}$$

$$= \begin{bmatrix} [k_{ii}] & [k_{ij}] & [k_{im}] \\ [k_{ji}] & [k_{jj}] & [k_{jm}] \\ [k_{im}] & [k_{jm}] & [k_{mm}] \end{bmatrix}$$

$$[k_{ij}] = [B_i]^T[D][B_j]$$

$$[k_{ji}] = [B_j]^T[D][B_i]$$

Stiffness Matrix

$$\int_{B^e} [B]^T [D][B] dV \{d\} - \int_{B^e} [N]^T \{f\} dV - \int_{\partial B^e} [N]^T \{\overline{T}\} dS = 0$$

$[k]^e$ $\{P\}^e$

$$[k]^e = \int_{B^e} [B]^T [D][B] dV$$

$$\{P\}^e = \int_{B^e} [N]^T \{f\} dV + \int_{\partial B^e} [N]^T \{\overline{T}\} dS$$

Body forces $\{P\}_b^e = \int_{B^e} [N]^T \{f\} dV$

$$[N] = \begin{bmatrix} N_i & 0 & N_j & 0 & N_m & 0 \\ 0 & N_i & 0 & N_j & 0 & N_m \end{bmatrix}$$

$$\{P\}_b^e = \int_{B^e} [N]^T \{f\} dV = t \int_{A^e} \begin{bmatrix} N_i & 0 \\ 0 & N_i \\ N_j & 0 \\ 0 & N_j \\ N_m & 0 \\ 0 & N_m \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} dA \Rightarrow$$

$$P_{bi1} = t \int_{A^e} N_i f_1 dA$$
$$P_{bi2} = t \int_{A^e} N_i f_2 dA$$
$$P_{bj1} = t \int_{A^e} N_j f_1 dA$$
$$P_{bj2} = t \int_{A^e} N_j f_2 dA$$
$$P_{bm1} = t \int_{A^e} N_m f_1 dA$$
$$P_{bm2} = t \int_{A^e} N_m f_2 dA$$

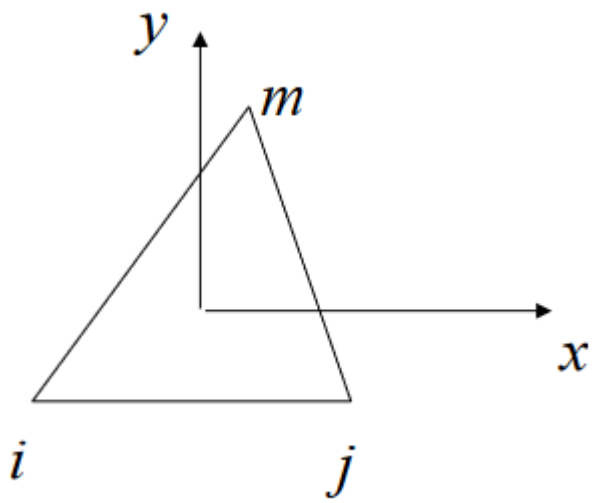
Body forces

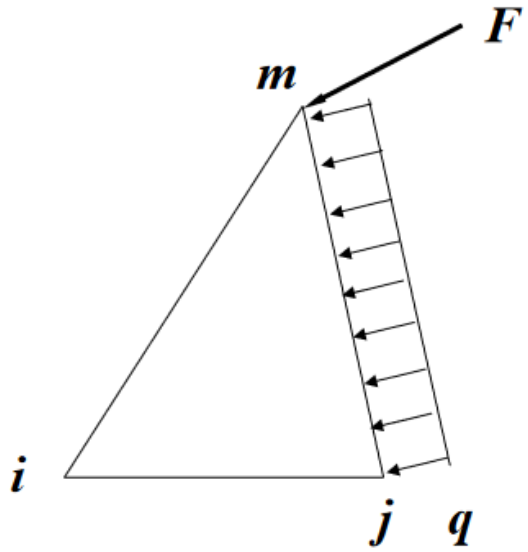
$$N_i = \frac{1}{2A}(\alpha_i + \beta_i x + \gamma_i y)$$

$$P_{bi1} = t\int_{A^e} N_i f_1 dA$$

$$P_{bi1} = \frac{t}{2A} f_1 \int_{A^e} (\alpha_i + \beta_i x + \gamma_i y) dA$$

$$= \frac{t}{2A} f_1 \left[\alpha_i \int_{A^e} dA + \beta_i \int_{A^e} x dA + \gamma_i \int_{A^e} y dA\right]$$





For $\boldsymbol{q}$: (N/m^2) $\quad \{P\}^e = \int_{\partial B^e} [N]^T \{\overline{T}\} dS$

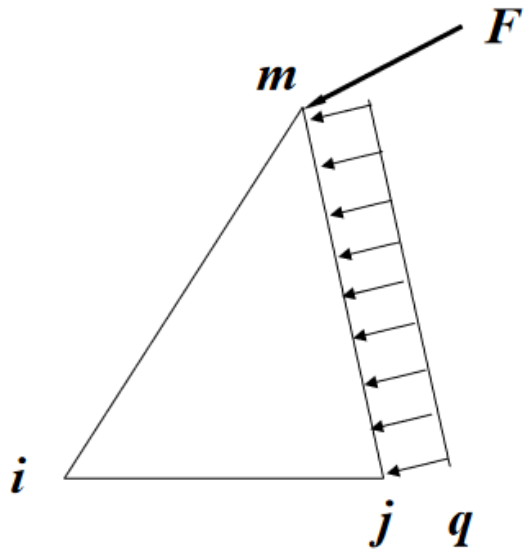
Here, ∂B^e is the edge, or a line.

Forces

On ∂B^e

$$N_i = 0 \qquad N_j = \frac{1}{2A}\left(\alpha_j + \beta_j x_j + \gamma_j y\right) = \frac{1}{2A}\left(\left(x_j - x_m\right)y_j + \left(x_m - x_i\right)y\right)$$

$$N_m = \frac{1}{2A}\left(\alpha_m + \beta_m x_m + \gamma_m y\right) = \frac{1}{2A}\left(\left(x_m - x_i\right)y_m + \left(x_j - x_i\right)y\right)$$



Surface forces

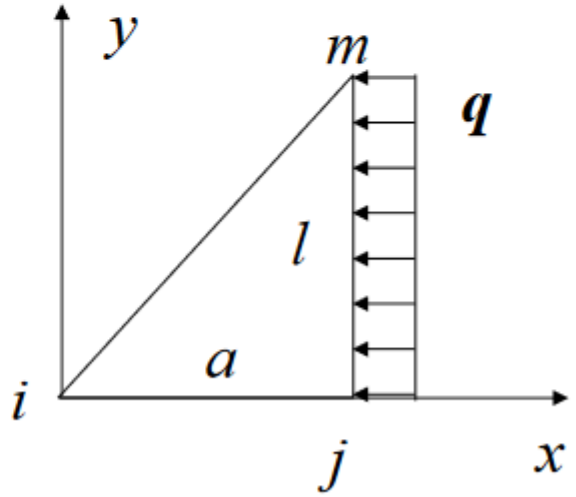
$$\{P\}^e = \int_{\{x=x_j, y=y\}} [N]^T \{\bar{T}\} dS = \int_{\{x=x_j, y=y\}} \begin{bmatrix} N_i & 0 \\ 0 & N_i \\ N_j & 0 \\ 0 & N_j \\ N_m & 0 \\ 0 & N_m \end{bmatrix} \begin{Bmatrix} -q \\ 0 \end{Bmatrix} dS = t \int_{y=y} \begin{Bmatrix} 0 \\ 0 \\ -N_j q \\ 0 \\ -N_m q \\ 0 \end{Bmatrix} dy$$

$$-\int_y N_j q dy = -\frac{q}{2A} \int_y \left[(x_j - x_m) y_j + (x_m - x_i) y \right] dy$$

$$= -\frac{q}{2A} \left[(x_j - x_m) y_j (y_m - y_j) + \frac{1}{2} (x_m - x_i)(y_m^2 - y_j^2) \right]$$

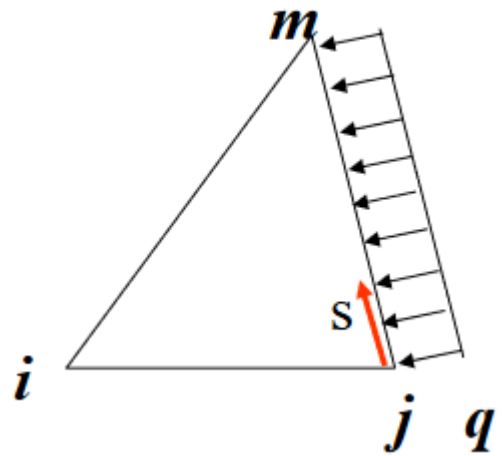
Note: $A = \frac{1}{2} al \qquad x_j - x_m = 0 \qquad x_m - x_i = a$

$$\rightarrow \qquad -\int_y N_j q dy = -\frac{ql}{2}$$



Forces

$$\{P\}^e = t\int_0^l \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \tilde{N}_j & 0 \\ 0 & \tilde{N}_j \\ \tilde{N}_m & 0 \\ 0 & \tilde{N}_m \end{bmatrix} \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} ds = t\int_0^l \begin{Bmatrix} 0 \\ 0 \\ \tilde{N}_j q_1 \\ \tilde{N}_j q_2 \\ \tilde{N}_m q_1 \\ \tilde{N}_m q_2 \end{Bmatrix} ds = t\int_0^l \begin{Bmatrix} 0 \\ 0 \\ \left(1-\frac{s}{l}\right)q_1 \\ \left(1-\frac{s}{l}\right)q_2 \\ \frac{s}{l}q_1 \\ \frac{s}{l}q_2 \end{Bmatrix} ds$$



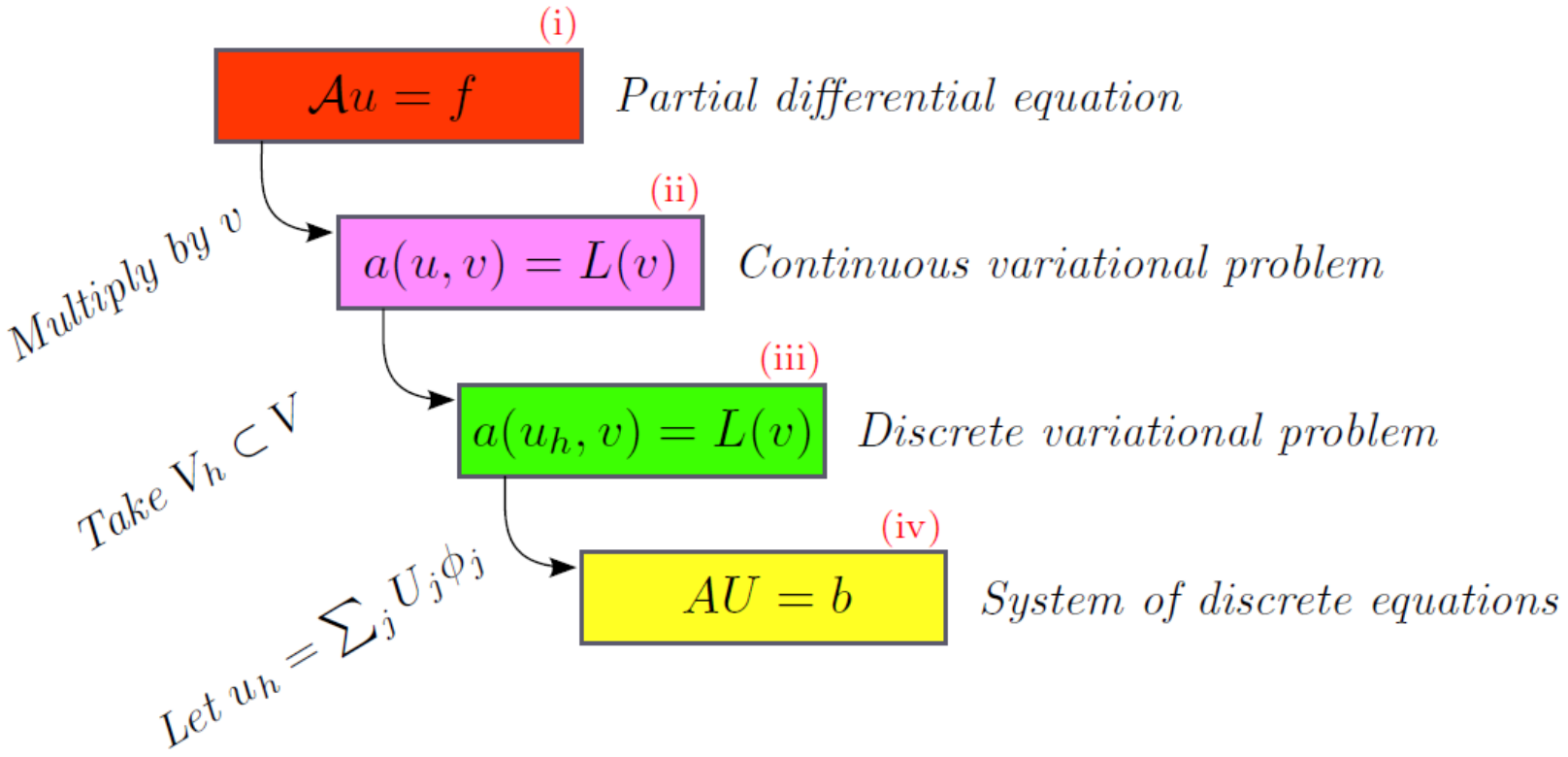
Finite Element Method: Quick Look

What is FEM

The finite element method is a framework and a recipe for discretization of differential equations

- Ordinary differential equations
- Partial differential equations
- Integral equations

- A recipe for discretization of PDE
- PDE $\rightarrow Ax = b$
- Different bases, stabilization, error control, adaptivity


(i)
$Au = f$
Partial differential equation
Multiply by v
(ii)
$a(u, v) = L(v)$
Continuous variational problem
Take $V_h \subset V$
(iii)
$a(u_h, v) = L(v)$
Discrete variational problem
Let $u_h = \sum_j U_j \phi_j$
(iv)
$AU = b$
System of discrete equations

$$-\Delta u = f \quad \text{in } \Omega$$
$$u = u_0 \quad \text{on } \partial\Omega$$

Poisson's equation arises in numerous applications:

- heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, water waves, magnetostatics, ...
- as part of numerical splitting strategies for more complicated systems of PDEs, in particular the Navier–Stokes equations

The simple recipe is: multiply the PDE by a test function v and integrate over Ω:

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

Then integrate by parts and set $v = 0$ on the Dirichlet boundary:

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s}_{=0}$$

We find that:

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}$

The trial space V and the test space $\hat{V}$ are (here) given by

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}$$

We approximate the continuous variational problem with a discrete variational problem posed on finite dimensional subspaces of V and $\hat{V}$:

$$V_h \subset V$$
$$\hat{V}_h \subset \hat{V}$$

Find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}_h \subset \hat{V}$

Choose a basis for the discrete function space:

$$V_h = \text{span}\,\{\phi_j\}_{j=1}^{N}$$

Make an ansatz for the discrete solution:

$$u_h = \sum_{j=1}^{N} U_j \phi_j$$

Test against the basis functions:

$$\int_\Omega \nabla(\underbrace{\sum_{j=1}^{N} U_j \phi_j}_{u_h}) \cdot \nabla \phi_i \,\mathrm{d}x = \int_\Omega f \phi_i \,\mathrm{d}x$$

Rearrange to get:

$$\sum_{j=1}^{N} U_j \underbrace{\int_\Omega \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}x}_{A_{ij}} = \underbrace{\int_\Omega f\phi_i \,\mathrm{d}x}_{b_i}$$

A linear system of equations:

$$AU = b$$

where

$$A_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}x$$

$$b_i = \int_\Omega f\phi_i \,\mathrm{d}x$$

(i) Partial differential equation:

$$\mathcal{A}u = f \quad \text{in } \Omega$$

(ii) Continuous variational problem: find $u \in V$ such that

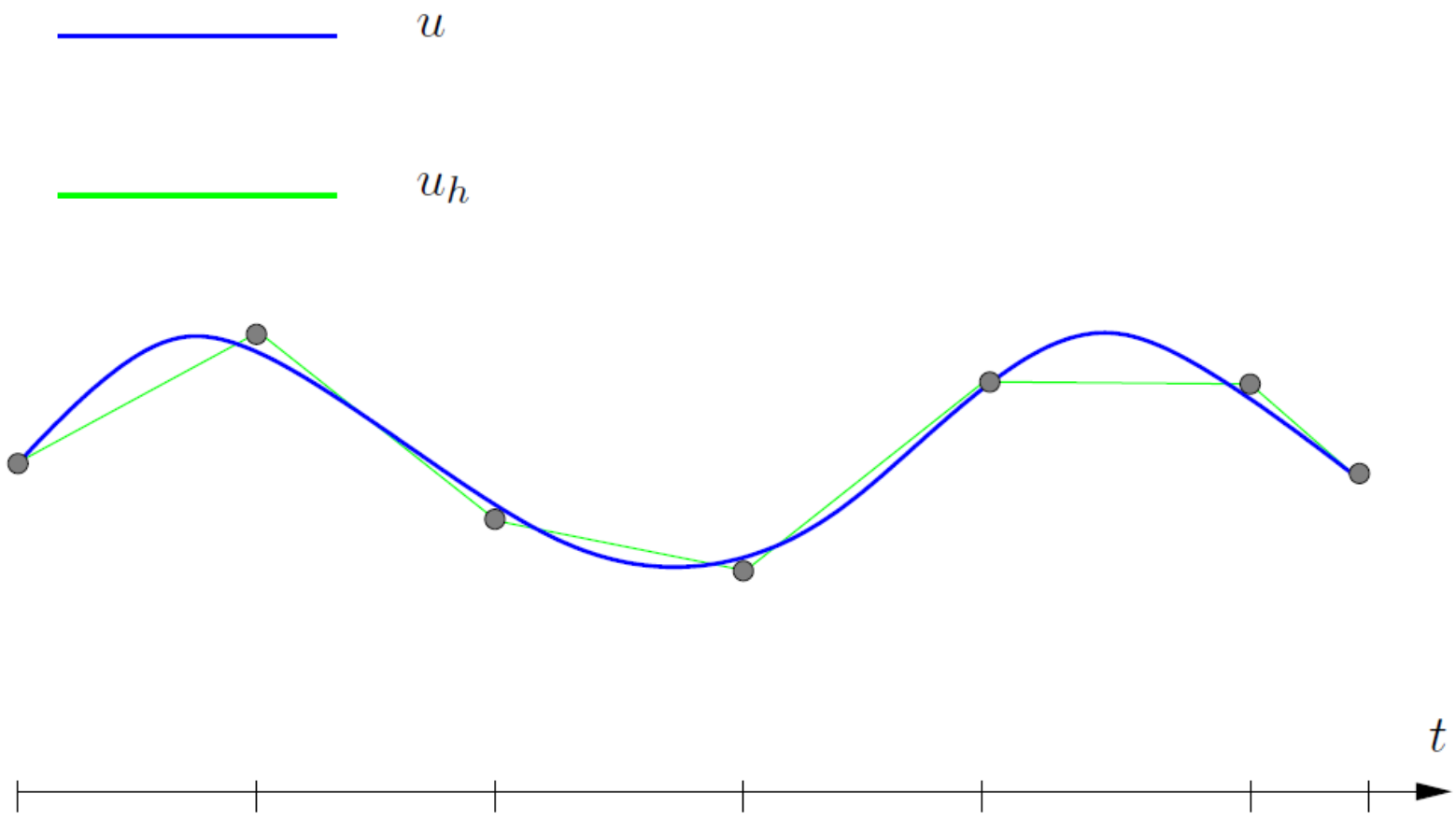
$$a(u, v) = L(v) \quad \text{for all } v \in \hat{V}$$

(iii) Discrete variational problem: find $u_h \in V_h \subset V$ such that

$$a(u_h, v) = L(v) \quad \text{for all } v \in \hat{V}_h$$

(iv) Discrete system of equations for $u_h = \sum_{j=1}^{N} U_j \phi_j$:

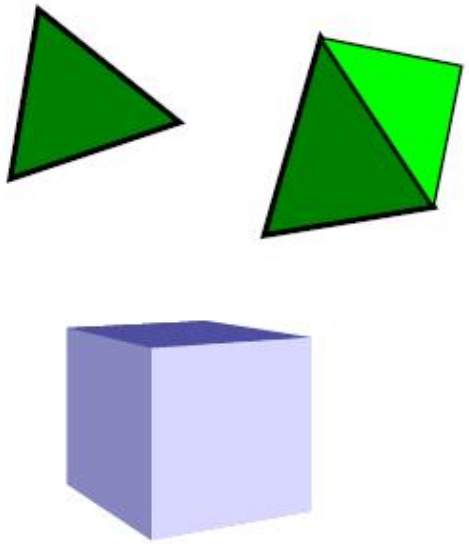
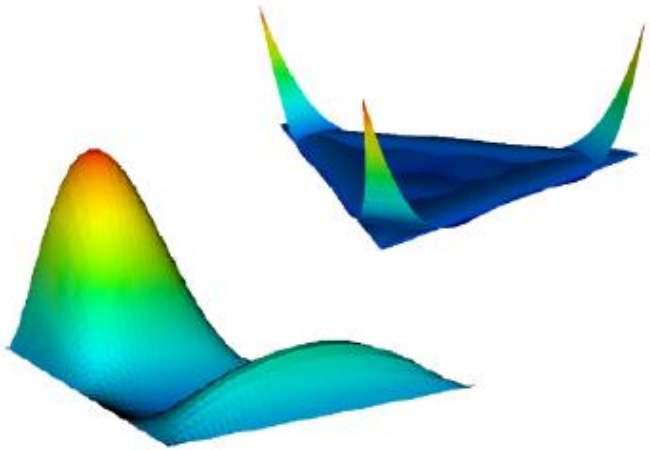
$$AU = b$$
$$A_{ij} = a(\phi_j, \phi_i)$$
$$b_i = L(\phi_i)$$


u
u_h
t

A finite element is a triple $(T, \mathcal{V}, \mathcal{L})$, where

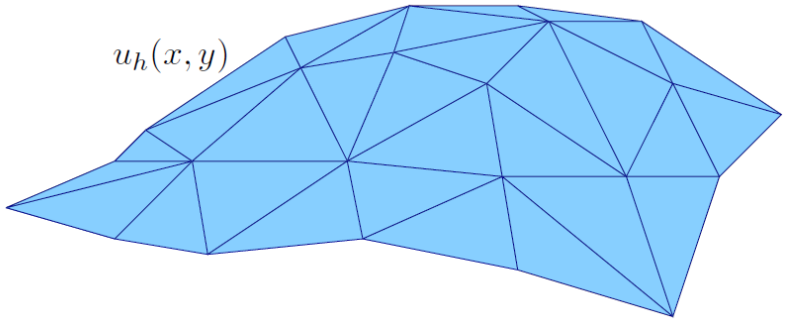
- the domain T is a bounded, closed subset of $\mathbb{R}^d$ (for $d = 1, 2, 3, \dots$) with nonempty interior and piecewise smooth boundary
- the space $\mathcal{V} = \mathcal{V}(T)$ is a finite dimensional function space on T of dimension n
- the set of degrees of freedom (nodes) $\mathcal{L} = \{\ell_1, \ell_2, \dots, \ell_n\}$ is a basis for the dual space $\mathcal{V}'$; that is, the space of bounded linear functionals on $\mathcal{V}$

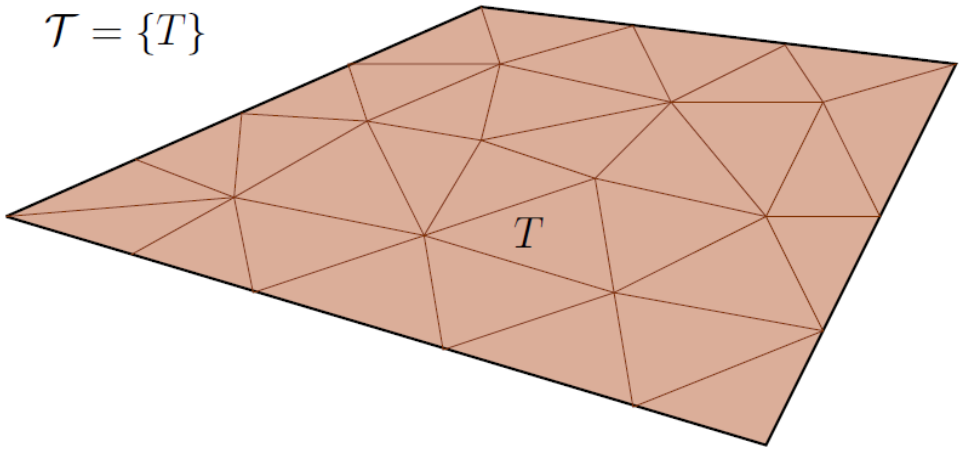
FEM: Definition

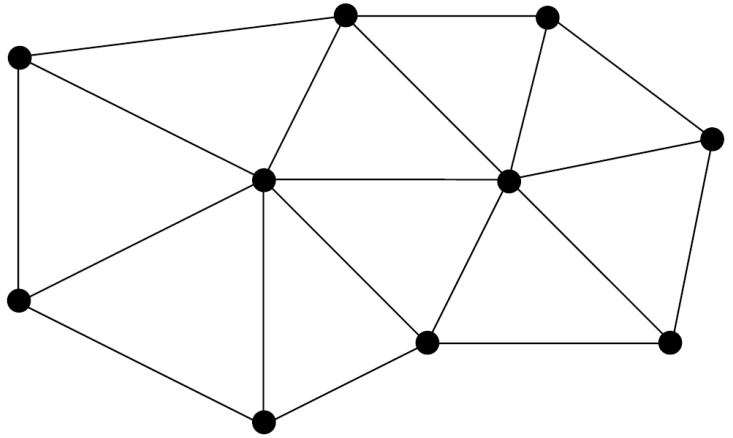
T	V	$\mathcal{L}$
		$v(\bar{x})$ $v(\bar{x}) \cdot n$ $\int_T v(x)w(x)\,\mathrm{d}x$

FEM: Linear Lagrange Element

- T is a line, triangle or tetrahedron
- $\mathcal{V}$ is the first-degree polynomials on T
- $\mathcal{L}$ is point evaluation at the vertices

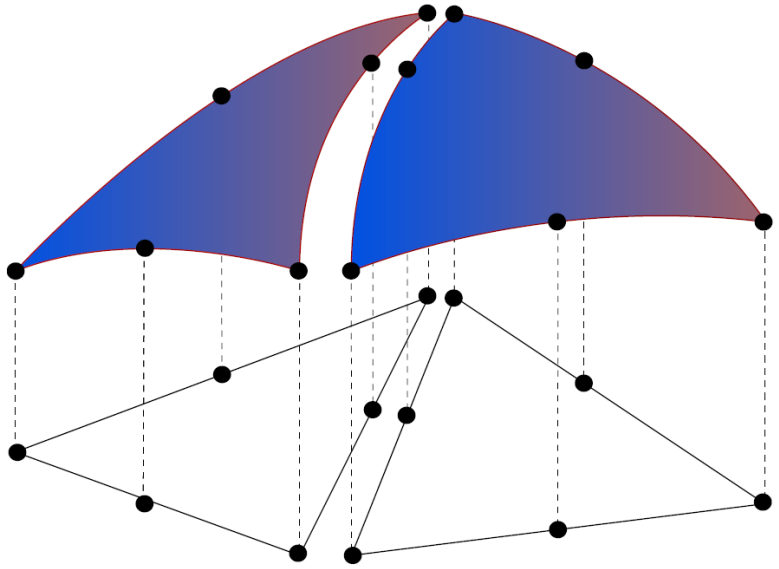


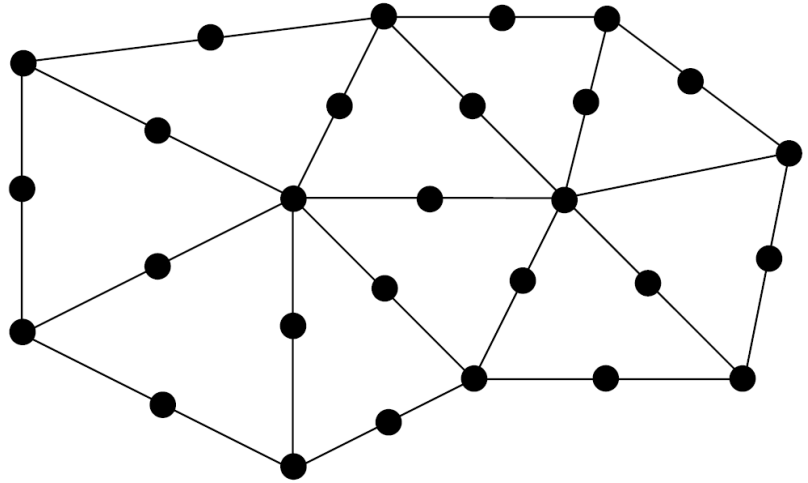




FEM: Quadratic Lagrange Element

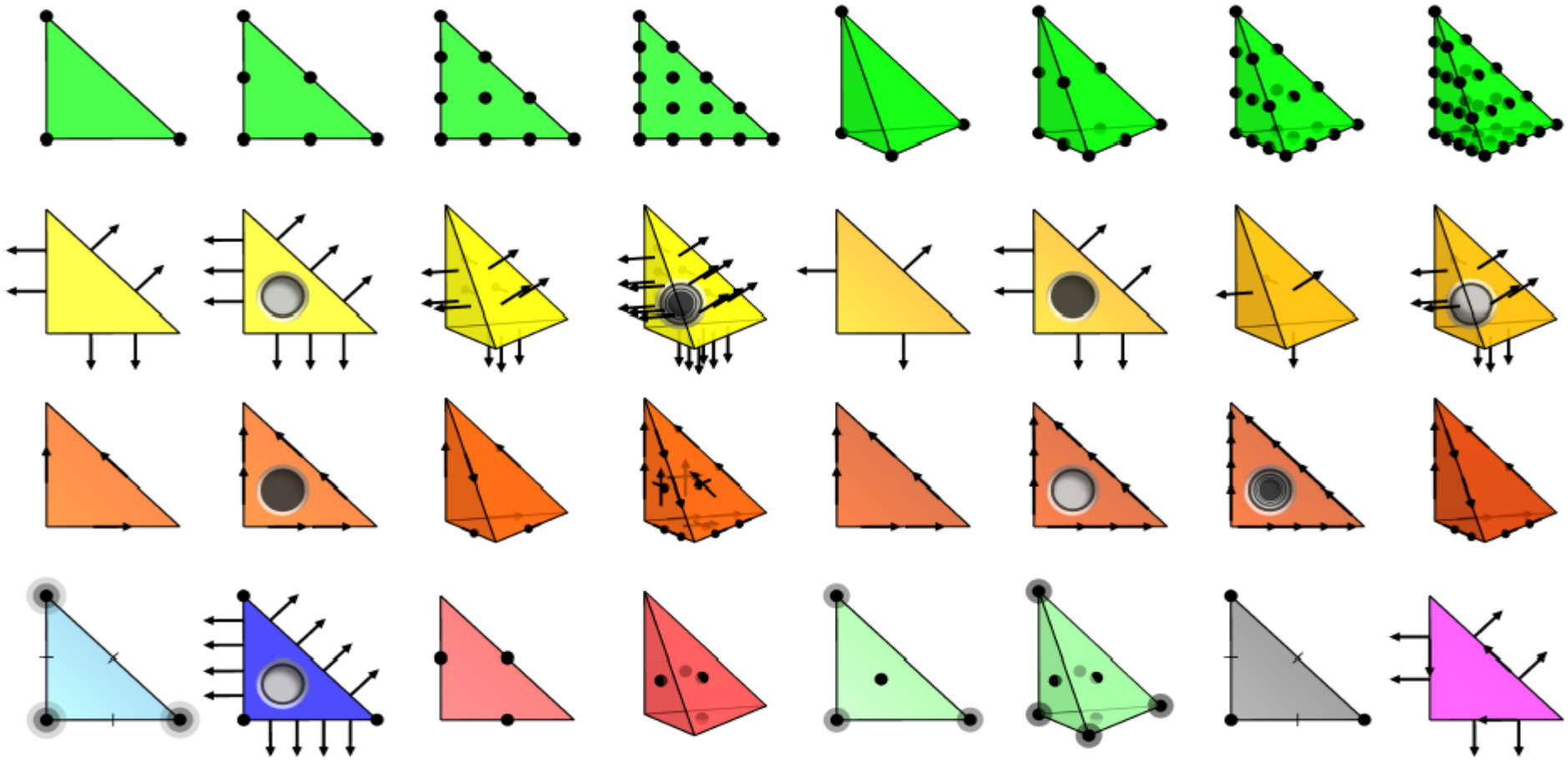
- T is a line, triangle or tetrahedron
- $\mathcal{V}$ is the second-degree polynomials on T
- $\mathcal{L}$ is point evaluation at the vertices and edge midpoints





FEM: Other Families of Elements

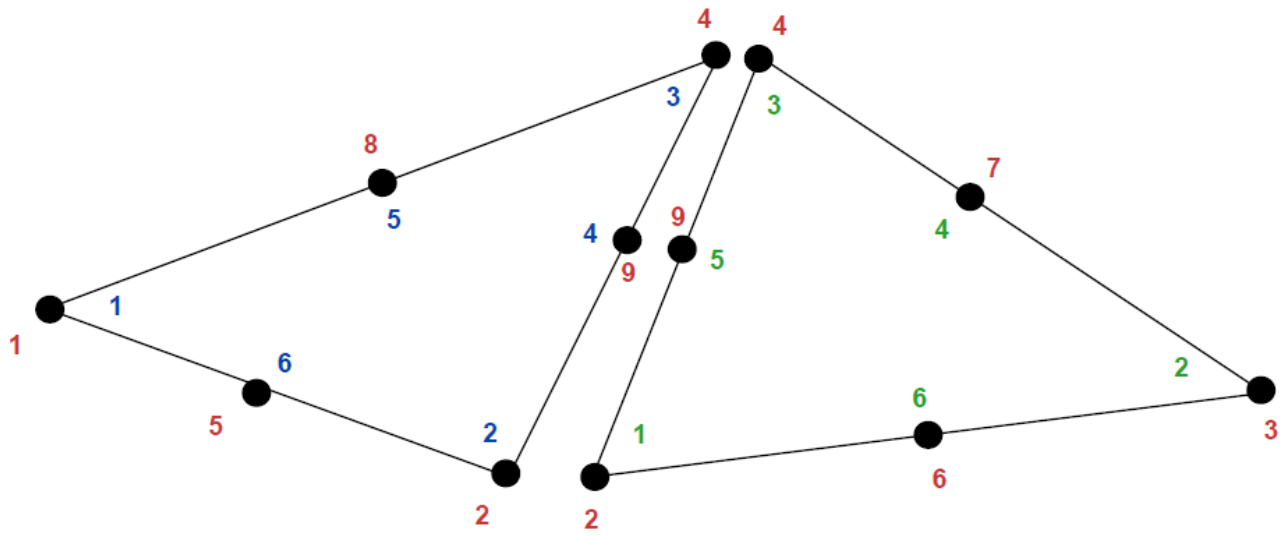
FEM: Other Families of Elements



The global matrix ι_T is defined by

$$I = \iota_T(i)$$

where I is the *global index* corresponding to the *local index* i



$A = 0$

for $T \in \mathcal{T}$

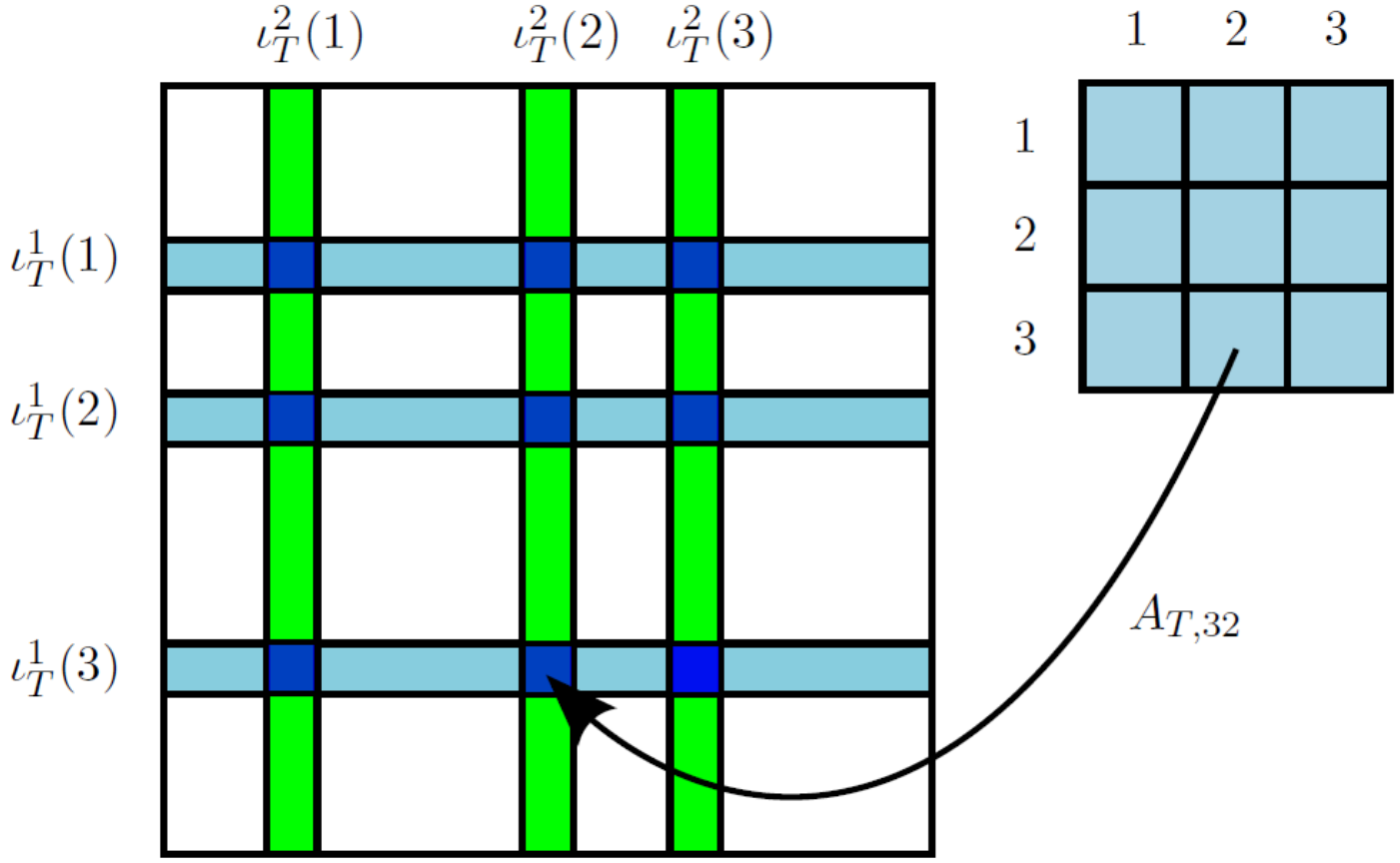
Compute the element matrix A_T

Compute the local-to-global mapping ι_T

Add A_T to A according to ι_T

end for

FEM: Assembling



FEniCS

- C++/Python Library
- 2003 Chicago
- Part of Debian and Ubuntu
- GNU Licence
- http://fenicsproject.org

Automated Solution of Differential Equations by the Finite Element Method

- started in 2003, collaboration between University of Chicago and Chalmers University of Technology
- end of 2011 - version 1.0, tutorial book published (jan 2012) with main contribution by 5 institutions (Simula Research Laboratory, University of Cambridge, University of Chicago, Texas Tech University, KTH Royal Institute of Technology)
- open source license (GNU LGPL v3)
- developement on launchpad (`https://launchpad.net/fenics-project`)

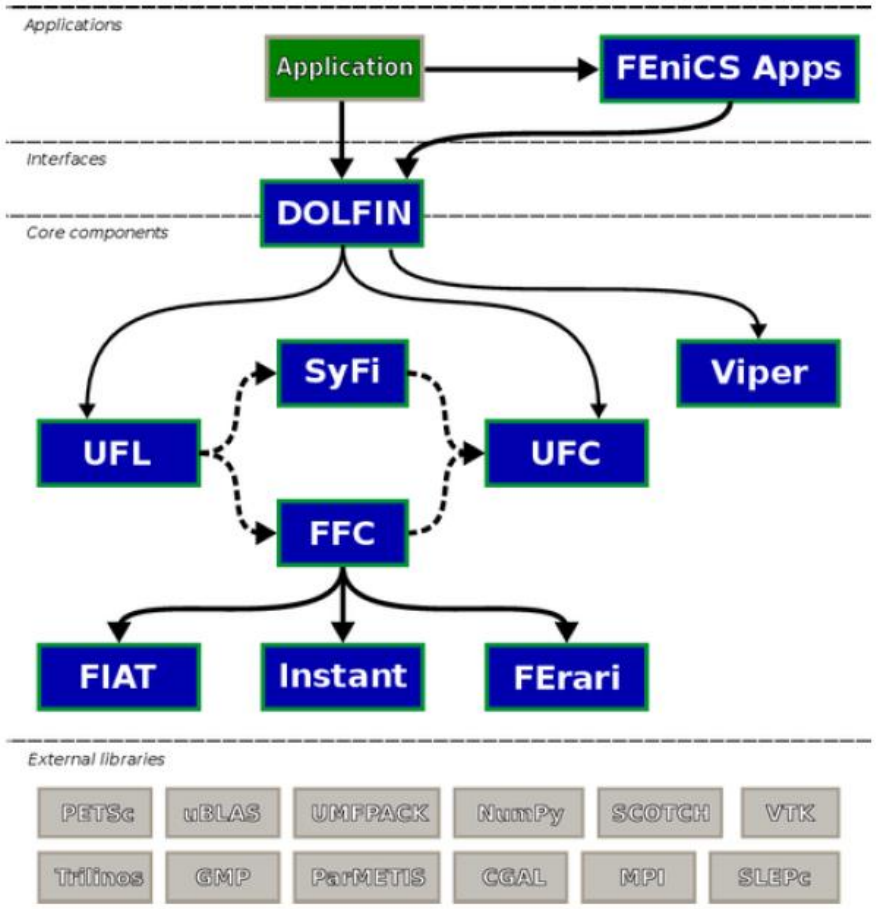
info:

- FEniCS book - Volume 84 of the Springer Lecture Notes in Computational Science and Engineering
- preliminary version of the book `https://launchpad.net/fenics-book`
- official tutorial `http://fenicsproject.org/documentation/tutorial`

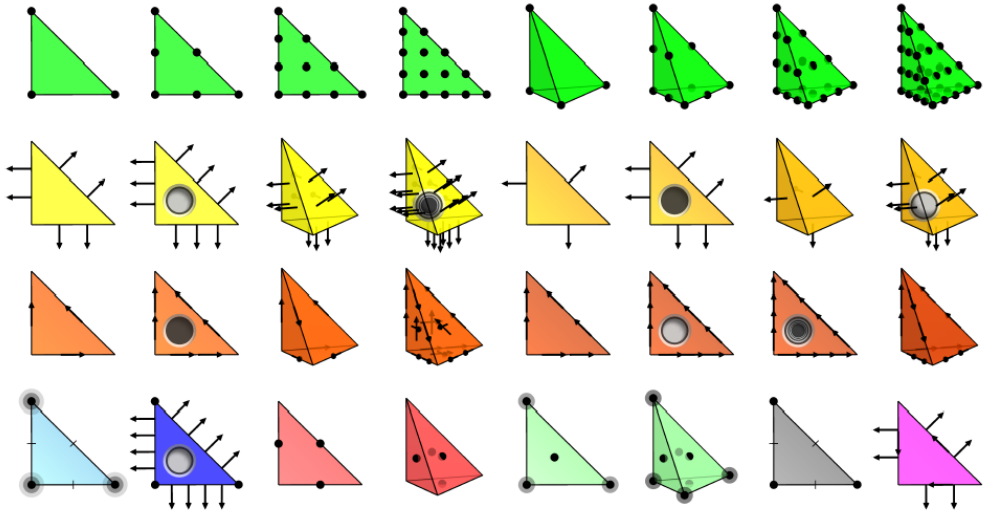
- core components:

Dolfin	C++/Python interface of FEniCS, providing a consistent Problem Solving Environment
FFC	FEniCS Form Compiler - compiler for multilinear forms by generating code (C++)
FIAT	FInite element Automatic Tabulator (curently Lagrange, mixed FE)
Instant	Python module that allows for instant inlining of C and C++ code in Python
UFC	Unified Form-assembly Code is a unified framework for finite element assembly
UFL	Unified Form Language is specific language for declaration of finite element discretizations of variational forms

- additional libraries: Dorsal, FErari, SyFi, Viper
- external libraries: PETSc, UMFPACK, Trilinos, CGAL, VTK,....


Applications
Application
FEniCS Apps
Interfaces
DOLFIN
Core components
SyFi
Viper
UFL
UFC
FFC
FIAT
Instant
FErari
External libraries
PETSc
uBLAS
UMFPACK
NumPy
SCOTCH
VTK
Trilinos
GMP
ParMETIS
CGAL
MPI
SLEPc

- Automated generation of basis functions
- Automated evaluation of variational forms
- Automated finite element assembly
- Automated adaptive error control



Collaborators

- Simula Research Laboratory
- University of Cambridge
- University of Chicago
- Texas Tech University
- KTH Royal Institute of Technology
- Chalmers University of Technology
- Imperial College London
- University of Oxford
- Charles University in Prague

Simple Example

$$\begin{cases} -\Delta u(x,y) = f(x,y) & \text{in } \Omega = [0,1]^2 \\ u(x,y) = u_B(x,y) & \text{on } \Gamma_D \\ -\dfrac{\partial u}{\partial \mathbf{n}} = g(x,y) & \text{on } \Gamma_N \end{cases}$$

where $\Gamma_D = \{(x,y) \in \Omega : x \in \{0,1\}\}$, $\Gamma_N = \{(x,y) \in \Omega : y \in \{0,1\}\}$, $f(x,y) = -6$, $u_B(x,y) = 1 + x + 2y^2$ and $g(x,y) = -4y$. The exact solution is given by

$$u(x,y) = 1 + x^2 + 2y^2$$

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} \qquad L(v) = \int_\Omega fv \, d\mathbf{x} - \int_{\Gamma_N} gv \, ds.$$

```
from dolfin import *                    # import the software library
set_log_level(PROGRESS)                 # suppress some outputs

# Create mesh and define function space
Th = UnitSquareMesh(15,15)              # regular mesh over [0,1]^2
Vh = FunctionSpace(Th, "CG", 1) # piecewise linear basis functions

# Define variational problem
g = Expression("-4*x[1]")               # Neumann boundary condition (x[1] is y)
f = Constant(-6)                        # right-hand side

u = TrialFunction(Vh)                   # find u s.t.
v = TestFunction(Vh)                    # for all v
a = inner(grad(u), grad(v))*dx  # bilinear form
L = f*v*dx - g*v*ds                     # linear form
```

```
# Define boundary conditions
def boundary(x, on_boundary):    # python function: identify boundary
    tol = 1E-14
    return on_boundary and (abs(x[0]) < tol or abs(x[0] - 1) < tol)

uB = Expression("1 + x[0] + 2*x[1]*x[1]")    # Dirichlet boundary condition
bc = DirichletBC(Vh, uB, boundary)           # set a DirichletBC object

# Compute solution
u = Function(Vh) # the solution is a function

problem = LinearVariationalProblem(a, L, u, bc) # set problem
solver = LinearVariationalSolver(problem)       # create solver
solver.parameters["linear_solver"] = "gmres"    # set solver
solver.parameters["preconditioner"] = "ilu"     # set preconditioner
gmres_prm = solver.parameters["krylov_solver"]  # set some parameters
gmres_prm["absolute_tolerance"] = 1e-7
gmres_prm["relative_tolerance"] = 1e-4
gmres_prm["maximum_iterations"] = 1000
solver.solve()                                  # solve
```

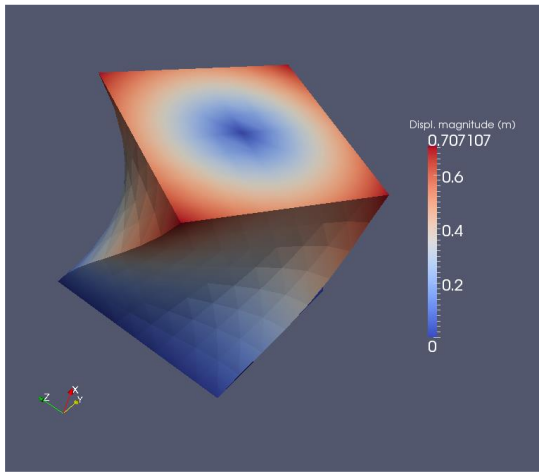
```
# L2 error and H1 error
uexact = Expression("1+x[0]*x[0]+2*x[1]*x[1]")
L2 = errornorm(uexact, u, norm_type="L2")
H1 = L2 + errornorm(uexact, u, norm_type="H10")
print "\nL2 error:", L2
print "\nH1 error:", H1, "\n"

# Plot solution and gradient
grad_u = project(grad(u), VectorFunctionSpace(Th, "CG", 1))
plot(u, axes=True)
plot(grad_u)

# Dump solution to file in VTK format (or XML format)
file = File("mixedPoisson.pvd")
file << u
file = File("mixedPoisson.xml")
file << u

# Hold plot
interactive()
```

```
from fenics import *
mesh = UnitCubeMesh (24 , 16 , 16)
V = VectorFunctionSpace (mesh , " Lagrange ", 1)
left = CompiledSubDomain ("( std :: abs(x[0])
        < DOLFIN_EPS ) && on_boundary ")
right = CompiledSubDomain ("(std :: abs(x[0] - 1.0)
        < DOLFIN_EPS ) && on_boundary ")
c = Expression (("0.0", "0.0", "0.0"), degree =0)
r = Expression (("0.0",
"0.5*( y0 +(x[1]-y0)*cos(t) -(x[2]-z0)*sin(t)-x[1])",
"0.5*( z0 +(x[1]-y0)*sin(t)+(x[2]-z0)*cos(t)-x[2])"),
y0=0.5, z0=0.5, t=pi/3, degree =3)
bcl = DirichletBC (V, c, left )
bcr = DirichletBC (V, r, right )
bcs = [bcl , bcr ]
v = TestFunction (V)
u = Function (V)
```




```
B = Constant ((0.0, -0.5, 0.0))
T = Constant ((0.1, 0.0, 0.0))
I = Identity (V. cell ().d)
F = I + grad (u)
Ic = tr(F.T*F)
J = det(F)
E, nu = 10.0, 0.3
mu , lmbda = Constant (E/(2*(1 + nu))),
        Constant (E*nu/((1 + nu)*(1 - 2*nu)))
psi = (mu/2)*(Ic - 3) - mu*ln(J) +
        ( lmbda /2)*(ln(J))**2
Pi = psi *dx - dot(B, u)*dx - dot(T, u)*ds
F = derivative (Pi , u, v)
solve (F == 0, u, bcs)
plot (u, interactive =True , mode ="
displacement ")
```

Poisson's equation

$$-\Delta u = f \quad \text{in } \Omega$$
$$u = 0 \quad \text{on } \partial\Omega$$

Finite element formulation

Find $u \in V$ such that

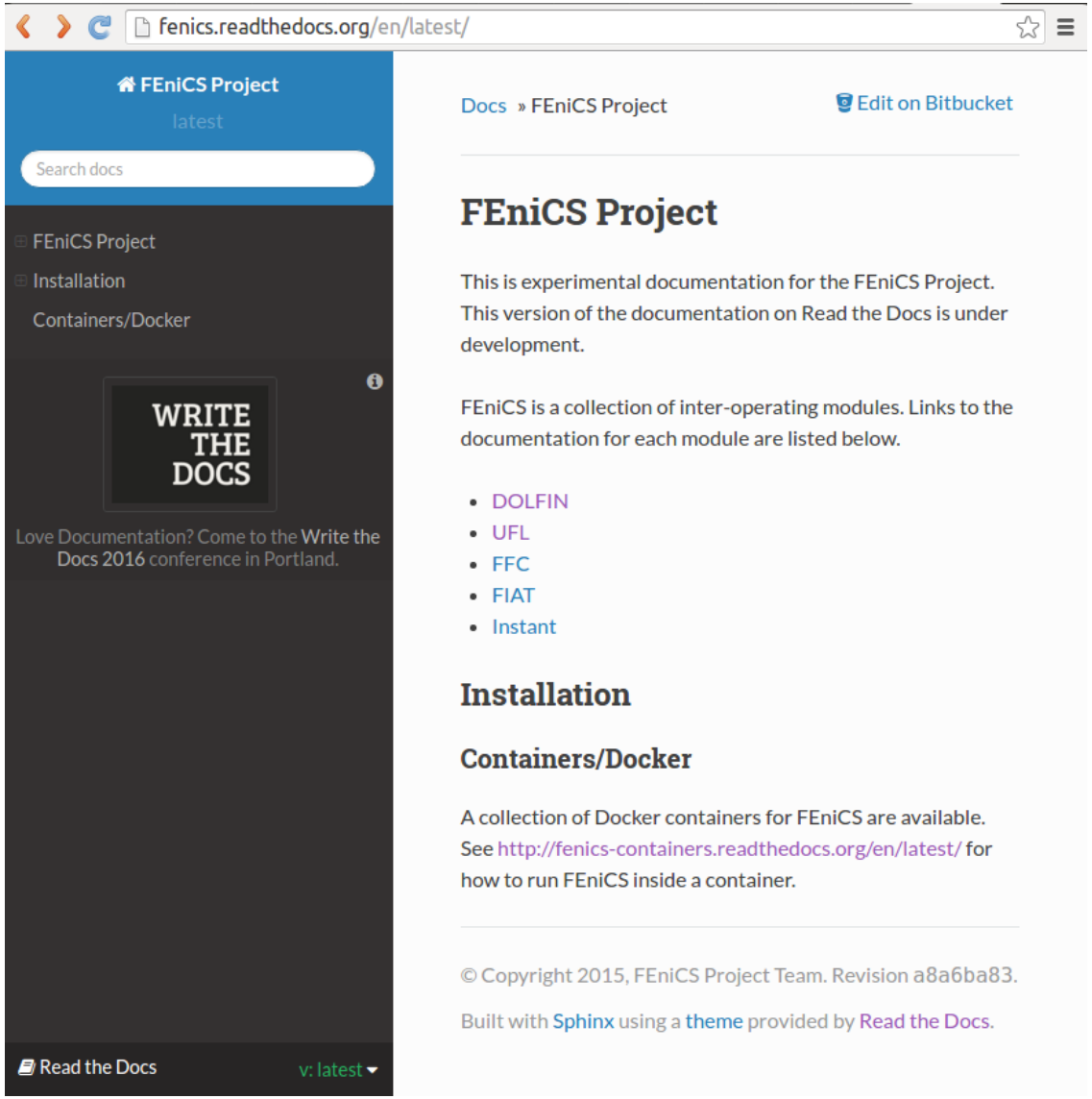
$$\underbrace{\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x}_{a(u,v)} = \underbrace{\int_\Omega f\, v \, \mathrm{d}x}_{L(v)} \quad \forall\, v \in V$$

```
from fenics import *
mesh = UnitSquareMesh (32 , 32)
V = FunctionSpace (mesh , " Lagrange ", 1)
u = TrialFunction (V)
v = TestFunction (V)
f = Expression ("x[0]*x[1]", degree =2)
a = dot( grad (u), grad (v))*dx
L = f*v*dx
bc = DirichletBC (V, 0.0, DomainBoundary ())
u = Function (V)
solve (a == L, u, bc)
plot (u)
```

FEniCS: Implementation

- `Mesh`, `Vertex`, `Edge`, `Face`, `Facet`, `Cell`
- `FiniteElement`, `FunctionSpace`
- `TrialFunction`, `TestFunction`, `Function`
- `grad()`, `curl()`, `div()`, ...
- `Matrix`, `Vector`, `KrylovSolver`, `LUSolver`
- `assemble()`, `solve()`, `plot()`

FEniCS: Documentation



Ubuntu

```
sudo add-apt-repository ppa:fenics-packages/fenics
sudo apt-get update
sudo apt-get install --no-install-recommends fenics
sudo apt-get dist-upgrade
```

Source

```
git clone git@bitbucket.org:fenics-project/fiat
git clone git@bitbucket.org:fenics-project/instant
git clone git@bitbucket.org:fenics-project/dijitso
git clone git@bitbucket.org:fenics-project/ufl
git clone git@bitbucket.org:fenics-project/ffc
git clone git@bitbucket.org:fenics-project/dolfin
git clone git@bitbucket.org:fenics-project/mshr
cd fiat    && pip3 install .
cd instant && pip3 install .
cd dijitso && pip3 install .
cd ufl     && pip3 install .
cd ffc     && pip3 install .
cd dolfin  && mkdir build && cd build && cmake .. && make install
cd mshr    && mkdir build && cd build && cmake .. && make install
```

- Identify
  - PDE
  - Boundary Condition
- Formulate PDE as variational problem
- Python program
  - Formulated variational problem is coded
  - Definitions of input data, f, u_0, Ω
- Solving variational problem
- Computing derived quantities
  - ∇u
- Visualize the results

$$
\begin{aligned}
-\Delta u &= f \quad \text{in } \Omega \\
u &= u_0 \quad \text{on } \partial\Omega
\end{aligned}
$$

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s}_{=0}$$

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

Variational Problem

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}$

The trial space V and the test space $\hat{V}$ are (here) given by

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \partial\Omega\}$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}$$

Variational Problem: Discrete Form

We approximate the continuous variational problem with a discrete variational problem posed on finite dimensional subspaces of V and $\hat{V}$:

$$V_h \subset V$$
$$\hat{V}_h \subset \hat{V}$$

Find $u_h \in V_h \subset V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x$$

for all $v \in \hat{V}_h \subset \hat{V}$

Variational Problem: Canonical Form

The following canonical notation is used in FEniCS: find $u \in V$ such that

$$a(u, v) = L(v)$$

for all $v \in \hat{V}$

For Poisson's equation, we have

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x$$

$$L(v) = \int_\Omega f v \, \mathrm{d}x$$

$a(u, v)$ is a *bilinear form* and $L(v)$ is a *linear form*

Test Problem

$$u(x, y) = 1 + x^2 + 2y^2$$

$$f = -\Delta u = -\Delta(1 + x^2 + 2y^2) = -(2 + 4) = -6$$

Step by Step Procedure in FEniCS

```
from fenics import *
mesh = UnitSquareMesh (8, 8)
V = FunctionSpace (mesh , " Lagrange ", 1)
u0 = Expression ("1 + x[0]*x[0] + 2*x[1]*x[1]",degree =2)
bc = DirichletBC (V, u0 , " on_boundary ")
f = Constant (-6.0)
u = TrialFunction (V)
v = TestFunction (V)
a = inner ( grad (u), grad (v))*dx
L = f*v*dx
u = Function (V)
solve (a == L, u, bc)
plot (u)
interactive () # If using VTK plotting
```

```
from fenics import *
```

- Import key classes
  - `UnitSquareMesh`
  - `FunctionSpace`
  - `Function`
  - …
- From the FEniCS user interface (DOLFIN)

Step by Step Procedure in FEniCS: Mesh

```
mesh = UnitSquareMesh (8, 8)
```

- This denes a mesh of $8 \times 8 \times 2 = 128$ triangles of the unit square
- Other useful classes for creating built-in meshes
  - `UnitIntervalMesh`
  - `UnitCubeMesh`
  - `UnitCircleMesh`
  - `UnitSphereMesh`
  - `RectangleMesh`
  - `BoxMesh`

Step by Step Procedure in FEniCS: Mesh

More complex geometries can be built using Constructive Solid Geometry (CSG) through the FEniCS component mshr:

```
from mshr import *
r = Rectangle ( Point (0.5, 0.5), Point (1.5, 1.5))
c = Circle ( Point (1.0, 1.0), 0.2)
g = r - c
mesh = generate_mesh (g, 10)
```

Step by Step Procedure in FEniCS: Function Space

```
V = FunctionSpace (mesh , " Lagrange ", 1)
```

- The second argument species the type of element, while the third argument is the degree of the basis functions on the element
- Other type elements
  - "Discontinuous Lagrange"
  - "Brezzi-Douglas-Marini"
  - "Raviart-Thomas"
  - "Crouzeix-Raviart"
  - "Nedelec 1st kind H(curl)"
  - "Nedelec 2nd kind H(curl)"

Step by Step Procedure in FEniCS: Expression

```
u0 = Expression ("1 + x[0]*x[0] + 2*x[1]*x[1]",degree =2)
```

- The formula must be written in C++ syntax, and the polynomial degree must be specified.
- The Expression class is very flexible and can be used to create complex user-defined expressions
- For more information

```
from fenics import *
help ( Expression )
```

Step by Step Procedure in FEniCS: BCs

```
bc = DirichletBC (V, u0 , " on_boundary ")
```

- This boundary condition states that a function in the function space defined by V should be equal to u0 on the domain defined by "on_ boundary"
- Note that the above line does not yet apply the boundary condition to all functions in the function space

```
" on_boundary " # The entire boundary
```

Step by Step Procedure in FEniCS: RHS

- f = -6 may be defined as follows:

```
f = Expression ("-6.0", degree =0)
```

- Or

```
f = Constant (-6.0)
```

Step by Step Procedure in FEniCS: Variational Problem

- Trial and Test Functions

```
u = TrialFunction (V)
v = TestFunction (V)
```

- Bilinear form and Linear Form a(u,v) and L(v)

```
a = inner ( grad (u), grad (v))*dx
L = f*v*dx
```

Step by Step Procedure in FEniCS: Solving

- Solve Function

```
u = Function (V)
solve (a == L, u, bc)
```

Note: reuse of the variable name u as both a `TrialFunction` in the variational problem and a `Function` to store the solution.

Step by Step Procedure in FEniCS: Post-Processing

- For Post Processing
  - Paraview
  - Mayavi
  - Store in VTK format

```
file = File (" poisson . pvd")
file << u
plot (u)
interactive () # If using VTK plotting
```

Stokes Equation: Non-linear PDEs

$$
\begin{aligned}
-\operatorname{div}(2\nu\epsilon(u) - p\,\mathrm{I}) &= f \quad \text{in } \Omega \\
\operatorname{div} u &= 0 \quad \text{in } \Omega \\
\epsilon(u) &= \tfrac{1}{2}\left(\operatorname{grad} u + (\operatorname{grad} u)^T\right) \\
u &= 0 \quad \text{on } \partial\Omega_D \\
-(2\nu\epsilon - p\,\mathrm{I}) \cdot n &= p_0\, n \quad \text{on } \partial\Omega_N
\end{aligned}
$$

$$
\nu = \nu(u)
$$

Variational Form: Non-linear PDEs

Assume that $u \in V$ and $p \in Q$, then $w = (u, p) \in V \times Q = W$.
Let $f = 0$.

Step 1

Multiply by test functions $(v, q) \in W$ and integrate first equation by parts

$$\int_\Omega 2\nu\epsilon(u) \cdot \operatorname{grad} v \,\mathrm{d}x - \int_\Omega p \operatorname{div} v \,\mathrm{d}x - \int_{\partial\Omega} (2\nu\epsilon(u) - p\,\mathrm{I}) \cdot n \cdot v \,\mathrm{d}s = 0$$

$$\int_\Omega \operatorname{div} u \, q \,\mathrm{d}x = 0$$

Step 2

Adding the equations and incorporating the boundary conditions we obtain: find $(u, p) \in W = V_0 \times Q$ such that

$$\int_\Omega 2\nu\epsilon(u) \cdot \operatorname{grad} v \, \mathrm{d}x - \int_\Omega p \operatorname{div} v \, \mathrm{d}x - \int_\Omega \operatorname{div} u \, q \, \mathrm{d}x + \int_{\partial\Omega_N} p_0 \, v \cdot n \, \mathrm{d}s = 0$$

for all $(v, q) \in W = V_0 \times Q$ where $V_0 = \{v \in V \text{ such that } v|_{\partial\Omega_D} = 0\}$.

Variational Form: Non-linear PDEs

Find $w \in W$ such that

$$F(w; y) = 0$$

for all $y \in \hat{W}$.

The functions are $w = (u, p)$, $y = (v, q)$ and the form F is

$$F(w; y) = \int_\Omega 2\nu\epsilon(u) \cdot \operatorname{grad} v \, \mathrm{d}x - \int_\Omega p \operatorname{div} v \, \mathrm{d}x - \int_\Omega \operatorname{div} u \, q \, \mathrm{d}x + \int_{\partial\Omega_N} p_0 \, v \cdot n \, \mathrm{d}s$$

More concepts

- Mixed function spaces
- Integration over boundaries
- Solving nonlinear problems (if nonlinear viscosity)
- (Reading a mesh from file)
- (Adjusting parameters)

Read From File

DOLFIN can read and write meshes from its own `.xml` or `.xml.gz` format

```
mesh = Mesh("dolfin-1.xml.gz")
plot(mesh)
```

Conversion tools exist for other mesh formats

```
$ man dolfin-convert
```

We will need the normal on the mesh boundary facets:

```
n = FacetNormal(mesh)
```

Mixed Function Spaces

Mixed elements are created by taking the product of more basic elements

```
# Define Taylor--Hood function space W
P2 = VectorElement("Lagrange", triangle, 2)
P1 = FiniteElement("Lagrange", triangle, 1)
TH = P2 * P1
#TH = MixedElement([P2, P1])
W = FunctionSpace(mesh, TH)
```

Mixed Function Spaces

You can define functions on mixed spaces and split into components:

```
w = Function(W)
(u, p) = split(w)
```

... and arguments:

```
y = TestFunction(W)
(v, q) = split(y)
# (v, q) = TestFunctions(W)
```

Function vs Expression

Function

... is described by expansion coefficients with reference to a `FunctionSpace` with a given basis: $u = \sum_i u_i \phi_i$

```
u = Function(V) # Defines the function space
u.vector()      # The coefficients
```

Expression

... given by an evaluation formula (more or less explicit)

```
f = Expression("...", degree=...)

class Source(Expression):
    def eval(self, values, x)
        ...
```

Function vs Expression

During **assemble** an **Expression** is interpolated into a polynomial space of the given degree on each cell.

```
nu = 0.1
```

... or it can vary with the domain: $\nu = 1 + 100x_1$

```
nu = Expression("1 + 100*x[1]", degree=1)
```

... or it can vary with the unknown: $\nu = (u \cdot u)^{1/2}$

```
w = Function(W)
(u, p) = split(w)
def viscosity(u):
    return inner(u, u)**(1./2)
nu = viscosity(u)
```

BCs on Mixed Spaces

Assume that we have a mixed function space:

```
W = FunctionSpace(mesh, V * Q)
```

The subspaces of W can be retrieved using `sub`:

```
W0 = W.sub(0)
```

Note that `W0` is not completely the same as `V`

The following code defines a homogenous Dirichlet (boundary) condition on the first subspace at the part where $x_0 = 0$.

```
g = (0.0, 0.0)
bc = DirichletBC(W.sub(0), g, "near(x[0], 0.0)")
```

Stokes: Variational Form

Assume that we have

```
w = Function(W)
(u, p) = split(w)
(v, q) = TestFunctions(W)
p0 = ...; nu = ...; n = ...
```

We can now specify the linear form F

```
epsilon = 2*sym(grad(u))
F = (nu*inner(epsilon, grad(v)) \
    - div(u)*q - div(v)*p)*dx \
    + p0*dot(v, n)*ds
```

Note that `dx` denotes integration over cells, `ds` denotes integration over exterior (boundary) facets, `dS` denotes integration over interior facets.

Stokes: Solving

Once a variational problem has been defined, it may be solved by calling the `solve` function (as for linear problems):

```
solve(F == 0, w, bc)
```

Or more verbosely

```
dF = derivative(F, w)
pde = NonlinearVariationalProblem(F, w, bc, dF)
solver = NonlinearVariationalSolver(pde)
solver.solve()
```

Extracting the subfunctions (as DOLFIN functions)

```
(u, p) = w.split(deepcopy=True)
```

Parameters Adjustments

```
from fenics import *
info(parameters, True)
parameters["form_compiler"]["cpp_optimize"] = True
#parameters["form_compiler"]["optimize"] = True
```

```
solver = NonlinearVariationalSolver(pde)
info(solver.parameters, True)
solver.parameters["symmetric"] = True
solver.parameters["newton_solver"]["maximum_iterations"]
    = 100
```

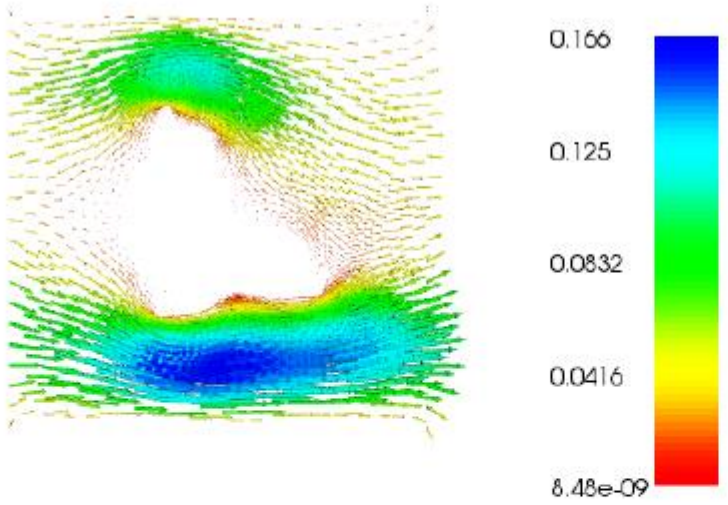
Complete Code

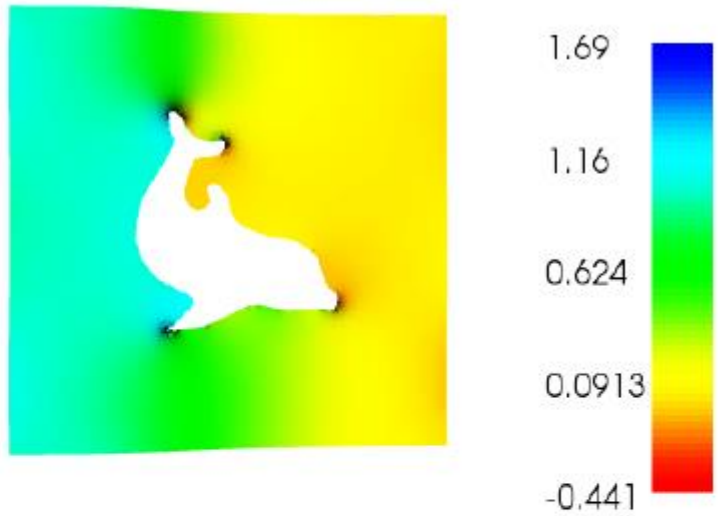
```
from __future__ import print_function
from fenics import *
# Use -02 optimization
parameters [" form_compiler "][" cpp_optimize "] = True
# Define mesh and geometry
mesh = Mesh (" dolfin -2. xml.gz")
x = SpatialCoordinate ( mesh )
n = FacetNormal ( mesh )
# Define Taylor -- Hood function space W
P2 = VectorElement (" Lagrange ", triangle , 2)
P1 = FiniteElement (" Lagrange ", triangle , 1)
TH = MixedElement ([P2 , P1 ])
W = FunctionSpace (mesh , TH)
# Define Function and TestFunction (s)
```

Complete Code

```
w = Function (W)
(u, p) = split (w)
(v, q) = TestFunctions (W)
# Define viscosity and bcs
nu = Expression ("0.2*(1+ pow(x[1],2))", degree =2)
p0 = (1.0 - x[0]) # or Expression ("1.0-x[0]", degree =1)
bcs = DirichletBC (W.sub (0), (0.0, 0.0),
" on_boundary && !( near (x[0], 0.0) || near (x[0], 1.0))")
# Define variational form
epsilon = sym ( grad (u))
F = (2*nu* inner ( epsilon , grad (v)) - div(u)*q - div(v)*p)*dx\
+ p0*dot(v,n)*ds
# Solve problem
solve (F == 0, w, bcs)
# Plot solutions
plot (u, title =" Velocity ")
plot (p, title =" Pressure ")
# interactive ()
```

Result






```
$ python stokes_example.py
```

Time Dependent PDE

$$\frac{\partial u}{\partial t} - \Delta u = f \quad \text{in } \Omega \text{ for } t > 0$$

$$u = g \quad \text{on } \partial\Omega \text{ for } t > 0$$

$$u = u^0 \text{ in } \Omega \text{ at } t = 0$$

$$\frac{\partial u}{\partial t}(t^n) \approx \frac{u^n - u^{n-1}}{\Delta t}, \quad u(t^n) \approx u^n, \quad f^n = f(t^n)$$

$$\frac{u^n - u^{n-1}}{\Delta t} - \Delta u^n = f^n$$

1. Start with u^0 and choose a timestep $\Delta t > 0$.
2. For $n = 1, 2, \ldots$, solve for u^n:

$$u^n - \Delta t \Delta u^n = u^{n-1} + \Delta t f^n$$

Find $u^n \in V^n$ such that

$$a(u^n, v) = L^n(v)$$

for all $v \in \hat{V}$ where

$$a(u, v) = \int_\Omega uv + \Delta t \nabla u \cdot \nabla v \, \mathrm{d}x$$

$$L^n(v) = \int_\Omega u^{n-1} v + \Delta t f^n v \, \mathrm{d}x$$

Time Stepping Algorithm

Define the boundary condition
Compute u^0 *as the projection of the given initial value*
Define the forms a *and* L
Assemble the matrix A *from the bilinear form* a
$t \leftarrow \Delta t$
while $t \leqslant T$ **do**
  Assemble the vector b *from the linear form* L
  Apply the boundary condition
  Solve the linear system $AU = b$ *for* U *and store in* u^1
  $t \leftarrow t + \Delta t$
  $u^0 \leftarrow u^1$ (get ready for next step)
end while

- Exact Solution

$$u = 1 + x^2 + \alpha y^2 + \beta t$$

- Insert u into heat equation

$$f = \dot{u} - \Delta u = \beta - 2 - 2\alpha$$

The initial condition is

$$u^0 = 1 + x^2 + \alpha y^2$$

```
alpha = 3; beta = 1.2
t = 0.0
g = Expression("1 + x[0]*x[0] + \
                alpha*x[1]*x[1] + beta*t",
                alpha=alpha, beta=beta, t=t,
                degree=2)
```

Then, we must explicitly update t and g:

```
t = 1.0
g.t = t
```

An alternative (robust) approach is to define t as a Constant:

```
t = Constant(0.0)
g = Expression("...", ..., t=t, ...)
t.assign(1.0)
# No need to update g itself
```

We need to project the initial value into V_h:

```
u0 = project(g, V)
```

We can also interpolate the initial value into V_h:

```
u0 = interpolate(g, V)
```

For linear problems, this code

```
solve(a == L, u, bcs)
```

is equivalent to this

```
# Assembling a bilinear form yields a matrix
A = assemble(a)
# Assembling a linear form yields a vector
b = assemble(L)

# Applying boundary condition info to system
for bc in bcs:
    bc.apply(A, b)

# Solve Ax = b
solve(A, u.vector(), b)
```

```
# Decide on a time step
dt = 0.3

# Create Functions for previous and current sol.s
u0 = project(g, V)
u1 = Function(V)

# Define the variational formulation
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(beta - 2 - 2*alpha)
a = u*v*dx + dt*inner(grad(u), grad(v))*dx
L = u0*v*dx + dt*f*v*dx

# Define the boundary condition
bc = DirichletBC(V, g, "on_boundary")

# Assemble only once, before time-stepping
A = assemble(a)
```

```
T = 2          # Set end time
t.assign(dt) # Solve on [0, dt] first

while t <= T:
    b = assemble(L)   # Assemble the rhs vector
    bc.apply(A, b)    # Apply boundary conditions

    # Solve linear system
    solve(A, u1.vector(), b)

    # Update time and previous solution
    t1 = float(t + dt)
    t.assign(t1)          # t := t1 + dt
    u0.assign(u1)         # u0 := u1
```

Introduction to Finite Element Method and FEniCS

Panchatcharam Mariappan
Assistant Professor

IIT Tirupati

Navier-Stokes in 60 Lines

FEniCS: 3D Poisson Equation

```
1) from fenics import *
2) import numpy as np
3) T = 10.0             # final time
4) num_steps = 500      # number of time steps
5) dt = T / num_steps # time step size
6) mu = 1               # kinematic viscosity
7) rho = 1              # density
8) mesh = UnitSquareMesh(16, 16)
9) V = VectorFunctionSpace(mesh, 'P', 2)
10)Q = FunctionSpace(mesh, 'P', 1)
11)inflow  = 'near(x[0], 0)'
12)outflow = 'near(x[0], 1)'
13)walls   = 'near(x[1], 0) || near(x[1], 1)'
```

```
14)bcu_noslip = DirichletBC(V, Constant((0, 0)), walls)
15)bcp_inflow = DirichletBC(Q, Constant(8), inflow)
16)bcp_outflow = DirichletBC(Q, Constant(0), outflow)
17)bcu = [bcu_noslip]
18)bcp = [bcp_inflow, bcp_outflow]
19)u = TrialFunction(V)
20)v = TestFunction(V)
21)p = TrialFunction(Q)
22)q = TestFunction(Q)
23)u_n = Function(V)
24)u_  = Function(V)
25)p_n = Function(Q)
26)p_  = Function(Q)
27)U   = 0.5*(u_n + u)
28)n   = FacetNormal(mesh)
29)f   = Constant((0, 0))
30)k   = Constant(dt)
31)mu  = Constant(mu)
32)rho = Constant(rho)
```

FEniCS: 3D Poisson Equation

```
33)def epsilon(u):
34)     return sym(nabla_grad(u))
35)def sigma(u, p):
36)     return 2*mu*epsilon(u) - p*Identity(len(u))
37)F1 = rho*dot((u - u_n) / k, v)*dx + rho*dot(dot(u_n, nabla_grad(u_n)),
   v)*dx + inner(sigma(U, p_n), epsilon(v))*dx + dot(p_n*n, v)*ds -
   dot(mu*nabla_grad(U)*n, v)*ds - rho*dot(f, v)*dx
38)a1 = lhs(F1)
39)L1 = rhs(F1)
40)a2 = dot(nabla_grad(p), nabla_grad(q))*dx
41)L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx - (1/k)*div(u_)*q*dx
42)a3 = dot(u, v)*dx
43)L3 = dot(u_, v)*dx - k*dot(nabla_grad(p_ - p_n), v)*dx
44)A1 = assemble(a1)
45)A2 = assemble(a2)
46)A3 = assemble(a3)
47)[bc.apply(A1) for bc in bcu]
48)[bc.apply(A2) for bc in bcp]
```

```
49)t = 0
50)for n in range(num_steps):
51)     t += dt
52)     b1 = assemble(L1)
53)     [bc.apply(b1) for bc in bcu]
54)     solve(A1, u_.vector(), b1)
55)     b2 = assemble(L2)
56)     [bc.apply(b2) for bc in bcp]
57)     solve(A2, p_.vector(), b2)
58)     b3 = assemble(L3)
59)     solve(A3, u_.vector(), b3)
60)     plot(u_)
61)     u_e = Expression(('4*x[1]*(1.0 - x[1])', '0'), degree=2)
62)     u_e = interpolate(u_e, V)
63)     error = np.abs(u_e.vector().array() - u_.vector().array()).max()
64)     print('t = %.2f: error = %.3g' % (t, error))
65)     print('max u:', u_.vector().array().max())
66)     u_n.assign(u_)
67)     p_n.assign(p_)
68)interactive()
```