Object-oriented scientific computing

Pras Pathmanathan

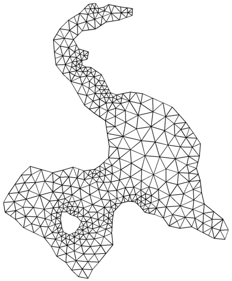
Summer 2012

The finite element method

Advantages of the FE method over the FD method

Main advantages of FE over FD

1. Deal with Neumann boundary conditions in a natural (systematic) way
2. Deals with irregular geometries much more easily



FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\boldsymbol{\nabla} u) \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

$$\begin{aligned} K_{jk} &= \int_\Omega \nabla \phi_j \cdot \nabla \phi_k \, \mathrm{d}V \\ b_j &= \int_\Omega f \phi_j \, \mathrm{d}V + \int_{\Gamma_2} g \phi_j \, \mathrm{d}S \end{aligned}$$

3. Alter linear system $KU = b$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1 \phi_1(\mathbf{x}) + \ldots U_N \phi_N(\mathbf{x})$

FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\nabla u)\cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

$$\begin{aligned} K_{jk} &= \int_\Omega \nabla\phi_j \cdot \nabla\phi_k \,\mathrm{d}V \\ b_j &= \int_\Omega f\phi_j \,\mathrm{d}V + \int_{\Gamma_2} g\phi_j \,\mathrm{d}S \end{aligned}$$

3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1\phi_1(\mathbf{x}) + \ldots U_N\phi_N(\mathbf{x})$

FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\nabla u) \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

$$\begin{aligned} K_{jk} &= \int_\Omega \nabla \phi_j \cdot \nabla \phi_k \, \mathrm{d}V \\ b_j &= \int_\Omega f \phi_j \, \mathrm{d}V + \int_{\Gamma_2} g \phi_j \, \mathrm{d}S \end{aligned}$$

3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1\phi_1(\mathbf{x}) + \ldots U_N\phi_N(\mathbf{x})$

FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\nabla u) \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

$$\begin{aligned} K_{jk} &= \int_\Omega \nabla \phi_j \cdot \nabla \phi_k \, \mathrm{d}V \\ b_j &= \int_\Omega f \phi_j \, \mathrm{d}V + \int_{\Gamma_2} g \phi_j \, \mathrm{d}S \end{aligned}$$

3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1\phi_1(\mathbf{x}) + \dots U_N\phi_N(\mathbf{x})$

FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\boldsymbol{\nabla} u) \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

$$\begin{aligned} K_{jk} &= \int_\Omega \boldsymbol{\nabla}\phi_j \cdot \boldsymbol{\nabla}\phi_k \, \mathrm{d}V \\ b_j &= \int_\Omega f\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S \end{aligned}$$

3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1\phi_1(\mathbf{x}) + \ldots U_N\phi_N(\mathbf{x})$

FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\boldsymbol{\nabla} u) \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

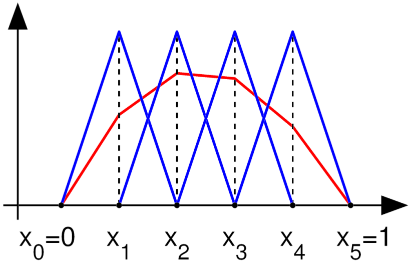
1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

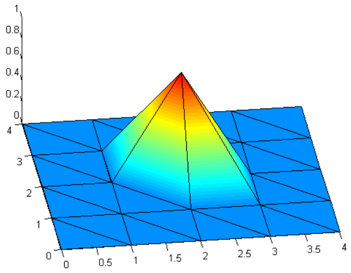
$$\begin{aligned} K_{jk} &= \int_\Omega \boldsymbol{\nabla}\phi_j \cdot \boldsymbol{\nabla}\phi_k \, \mathrm{d}V \\ b_j &= \int_\Omega f\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S \end{aligned}$$

3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1\phi_1(\mathbf{x}) + \ldots U_N\phi_N(\mathbf{x})$

Basis functions





Implementing Dirichlet boundary conditions

In practice, rather using the basis functions in $\mathcal{V}_0^h$ (i.e. bases satisfying $\phi_i = 0$ on Γ_1), we use $\mathcal{V}^h$, i.e. all the basis functions corresponding to all nodes in the mesh.

We then impose (any) Dirichlet boundary conditions by altering the appropriate rows of the linear system, for example, for $K\mathbf{U} = b$, if we want to impose $U_1 = c$

$$\begin{bmatrix} K_{11} & K_{12} & \dots & K_{1N} \\ K_{21} & K_{22} & \dots & K_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ K_{N1} & K_{N2} & \dots & K_{NN} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

Implementing Dirichlet boundary conditions

In practice, rather using the basis functions in $\mathcal{V}_0^h$ (i.e. bases satisfying $\phi_i = 0$ on Γ_1), we use $\mathcal{V}^h$, i.e. all the basis functions corresponding to all nodes in the mesh.

We then impose (any) Dirichlet boundary conditions by altering the appropriate rows of the linear system, for example, for $K\mathbf{U} = b$, if we want to impose $U_1 = c$

$$\begin{bmatrix} K_{11} & K_{12} & \dots & K_{1N} \\ K_{21} & K_{22} & \dots & K_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ K_{N1} & K_{N2} & \dots & K_{NN} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_N \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

Implementing Dirichlet boundary conditions

In practice, rather using the basis functions in $\mathcal{V}_0^h$ (i.e. bases satisfying $\phi_i = 0$ on Γ_1), we use $\mathcal{V}^h$, i.e. all the basis functions corresponding to all nodes in the mesh.

We then impose (any) Dirichlet boundary conditions by altering the appropriate rows of the linear system, for example, for $K\mathbf{U} = b$, if we want to impose $U_1 = c$

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ K_{21} & K_{22} & \dots & K_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ K_{N1} & K_{N2} & \dots & K_{NN} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_N \end{bmatrix} = \begin{bmatrix} c \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

Solving linear systems

Consider the general problem of solving the linear system

$$A\mathbf{x} = \mathbf{b}$$

where A is an $n \times n$ matrix and $\mathbf{b}$ an n-vector.

Direct solvers

- One approach is to compute A^{-1} and calculate $\mathbf{x} = A^{-1}\mathbf{b}$
- **Gaussian elimination** is an algorithm that is often used and is essentially equivalent to computing A^{-1}
- These approaches get too costly (in time and memory) for large n
- For $n < 10000$ (maybe even 50000) this may be the best approach

Solving linear systems

Consider the general problem of solving the linear system

$$A\mathbf{x} = \mathbf{b}$$

where A is an $n \times n$ matrix and $\mathbf{b}$ an n-vector.

Direct solvers

- One approach is to compute A^{-1} and calculate $\mathbf{x} = A^{-1}\mathbf{b}$
- **Gaussian elimination** is an algorithm that is often used and is essentially equivalent to computing A^{-1}
- These approaches get too costly (in time and memory) for large n
- For $n \leq 10000$ (maybe even 50000) this may be the best approach

Solving linear systems

Consider the general problem of solving the linear system

$$A\mathbf{x} = \mathbf{b}$$

where A is an $n \times n$ matrix and $\mathbf{b}$ an n-vector.

Direct solvers

- One approach is to compute A^{-1} and calculate $\mathbf{x} = A^{-1}\mathbf{b}$
- **Gaussian elimination** is an algorithm that is often used and is essentially equivalent to computing A^{-1}
- These approaches get too costly (in time and memory) for large n
- For $n \leq 10000$ (maybe even 50000) this may be the best approach

Solving linear systems

Consider the general problem of solving the linear system

$$A\mathbf{x} = \mathbf{b}$$

where A is an $n \times n$ matrix and $\mathbf{b}$ an n-vector.

Direct solvers

- One approach is to compute A^{-1} and calculate $\mathbf{x} = A^{-1}\mathbf{b}$
- **Gaussian elimination** is an algorithm that is often used and is essentially equivalent to computing A^{-1}
- These approaches get too costly (in time and memory) for large n
- For $n \leq 10000$ (maybe even 50000) this may be the best approach

Solving linear systems

Iterative solvers

- Often A is **sparse**
- This means that computing $A\mathbf{y}$ for given $\mathbf{y}$ is cheap.
- Choose initial guess $\mathbf{x}_0$
- An iterative solver takes in a current guess $\mathbf{x}_n$ and provides an improved solution $\mathbf{x}_{n+1}$, just using matrix-vector products
- Common iterative solvers are *conjugate gradients* (for when A is symmetric and positive-definite) and *GMRES*.

Preconditioning

- Question is then: (i) does $\mathbf{x}_n \to \mathbf{x}$ and (ii) how fast does $\mathbf{x}_n \to \mathbf{x}$?
- For any non-singular matrix P, the system

$$PA\mathbf{x} = P\mathbf{b}$$

  is equivalent to the original system $A\mathbf{x} = \mathbf{b}$.
- By choosing P appropriately, can obtain (massively) improved convergence—preconditioning

Solving linear systems

Iterative solvers

- Often A is **sparse**
- This means that computing $A\mathbf{y}$ for given $\mathbf{y}$ is cheap.
- Choose initial guess $\mathbf{x}_0$
- An iterative solver takes in a current guess $\mathbf{x}_n$ and provides an improved solution $\mathbf{x}_{n+1}$, just using matrix-vector products
- Common iterative solvers are *conjugate gradients* (for when A is symmetric and positive-definite) and *GMRES*.

Preconditioning

- Question is then: (i) does $\mathbf{x}_n \to \mathbf{x}$ and (ii) how fast does $\mathbf{x}_n \to \mathbf{x}$?
- For any non-singular matrix P, the system

$$PA\mathbf{x} = P\mathbf{b}$$

is equivalent to the original system $A\mathbf{x} = \mathbf{b}$.
- By choosing P appropriately, can obtain (massively) improved convergence—preconditioning.

Solving linear systems

Iterative solvers

- Often A is **sparse**
- This means that computing $A\mathbf{y}$ for given $\mathbf{y}$ is cheap.
- Choose initial guess $\mathbf{x}_0$
- An iterative solver takes in a current guess $\mathbf{x}_n$ and provides an improved solution $\mathbf{x}_{n+1}$, just using matrix-vector products
- Common iterative solvers are *conjugate gradients* (for when A is symmetric and positive-definite) and *GMRES*.

Preconditioning

- Question is then: (i) does $\mathbf{x}_n \to \mathbf{x}$ and (ii) how fast does $\mathbf{x}_n \to \mathbf{x}$?
- For any non-singular matrix P, the system

$$PA\mathbf{x} = P\mathbf{b}$$

  is equivalent to the original system $A\mathbf{x} = \mathbf{b}$.
- By choosing P appropriately, can obtain (massively) improved convergence—preconditioning.

Solving linear systems

Iterative solvers

- Often A is **sparse**
- This means that computing $A\mathbf{y}$ for given $\mathbf{y}$ is cheap.
- Choose initial guess $\mathbf{x}_0$
- An iterative solver takes in a current guess $\mathbf{x}_n$ and provides an improved solution $\mathbf{x}_{n+1}$, just using matrix-vector products
- Common iterative solvers are *conjugate gradients* (for when A is symmetric and positive-definite) and *GMRES*.

Preconditioning

- Question is then: (i) does $\mathbf{x}_n \to \mathbf{x}$ and (ii) how fast does $\mathbf{x}_n \to \mathbf{x}$?
- For any non-singular matrix P, the system

$$PA\mathbf{x} = P\mathbf{b}$$

  is equivalent to the original system $A\mathbf{x} = \mathbf{b}$.
- By choosing P appropriately, can obtain (massively) improved convergence—**preconditioning**

Solving linear systems

Iterative solvers

- Often A is **sparse**
- This means that computing $A\mathbf{y}$ for given $\mathbf{y}$ is cheap.
- Choose initial guess $\mathbf{x}_0$
- An iterative solver takes in a current guess $\mathbf{x}_n$ and provides an improved solution $\mathbf{x}_{n+1}$, just using matrix-vector products
- Common iterative solvers are *conjugate gradients* (for when A is symmetric and positive-definite) and *GMRES*.

Preconditioning

- Question is then: (i) does $\mathbf{x}_n \to \mathbf{x}$ and (ii) how fast does $\mathbf{x}_n \to \mathbf{x}$?
- For any non-singular matrix P, the system

$$PA\mathbf{x} = P\mathbf{b}$$

  is equivalent to the original system $A\mathbf{x} = \mathbf{b}$.
- By choosing P appropriately, can obtain (massively) improved convergence—**preconditioning**

Solving linear systems

Iterative solvers

- Often A is **sparse**
- This means that computing $A\mathbf{y}$ for given $\mathbf{y}$ is cheap.
- Choose initial guess $\mathbf{x}_0$
- An iterative solver takes in a current guess $\mathbf{x}_n$ and provides an improved solution $\mathbf{x}_{n+1}$, just using matrix-vector products
- Common iterative solvers are *conjugate gradients* (for when A is symmetric and positive-definite) and *GMRES*.

Preconditioning

- Question is then: (i) does $\mathbf{x}_n \to \mathbf{x}$ and (ii) how fast does $\mathbf{x}_n \to \mathbf{x}$?
- For any non-singular matrix P, the system

$$PA\mathbf{x} = P\mathbf{b}$$

  is equivalent to the original system $A\mathbf{x} = \mathbf{b}$.
- By choosing P appropriately, can obtain (massively) improved convergence—**preconditioning**

FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\boldsymbol{\nabla} u) \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

$$\begin{aligned} K_{jk} &= \int_\Omega \boldsymbol{\nabla}\phi_j \cdot \boldsymbol{\nabla}\phi_k \, \mathrm{d}V \\ b_j &= \int_\Omega f\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S \end{aligned}$$

3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1\phi_1(\mathbf{x}) + \ldots U_N\phi_N(\mathbf{x})$

The finite element method: assembly

Numerical quadrature

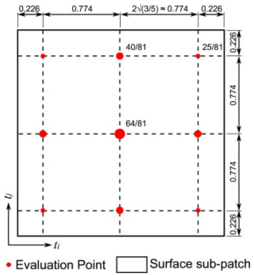
Suppose we want to compute

$$\int_{\text{unit square}} F(x, y)\,\mathrm{d}x\mathrm{d}y$$

We can use the approximation

$$\int_{\text{unit square}} F(x, y)\,\mathrm{d}x\mathrm{d}y \approx \sum_i w_i F(x_i, y_i)$$

where (x_i, y_i) are the *quadrature points*, and w_i the *weights*



Numerical quadrature

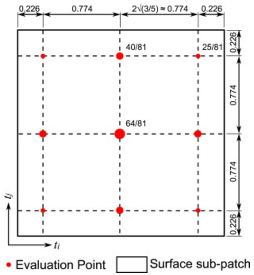
Suppose we want to compute

$$\int_{\text{unit square}} F(x,y)\,\mathrm{d}x\mathrm{d}y$$

We can use the approximation

$$\int_{\text{unit square}} F(x,y)\,\mathrm{d}x\mathrm{d}y \approx \sum_i w_i F(x_i, y_i)$$

where (x_i, y_i) are the *quadrature points*, and w_i the *weights*



Computing a finite element matrix/vector by *assembly*

Consider computing the mass matrix $M_{jk} = \int_\Omega \phi_j \phi_k \, \mathrm{d}V$, an N by N matrix say, and let's suppose (for clarity only) that we are in 2D.

We **do not** write out the full basis functions explicitly in computing this integral. Instead: firstly, we break the integral down into an integral over elements:

$$M_{jk} = \sum_{\mathcal{K}} \int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$$

Consider $\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$. **Key point**: The only basis functions with are non-zero in the triangle are the 3 basis functions corresponding to the 3 nodes of the element.

Therefore: compute the **elemental contribution to the mass matrix**, a 3 by 3 matrix of the form $\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$ for *3 choices of j and k only*.

Then **add** elemental contribution to full N by N mass matrix.

Computing a finite element matrix/vector by *assembly*

Consider computing the mass matrix $M_{jk} = \int_\Omega \phi_j \phi_k \,\mathrm{d}V$, an N by N matrix say, and let's suppose (for clarity only) that we are in 2D.

We **do not** write out the full basis functions explicitly in computing this integral. Instead: firstly, we break the integral down into an integral over elements:

$$M_{jk} = \sum_{\mathcal{K}} \int_{\mathcal{K}} \phi_j \phi_k \,\mathrm{d}V$$

Consider $\int_{\mathcal{K}} \phi_j \phi_k \,\mathrm{d}V$. **Key point**: The only basis functions with are non-zero in the triangle are the 3 basis functions corresponding to the 3 nodes of the element.

Therefore: compute the **elemental contribution to the mass matrix**, a 3 by 3 matrix of the form $\int_{\mathcal{K}} \phi_j \phi_k \,\mathrm{d}V$ for *3 choices of j and k only.*

Then **add** elemental contribution to full N by N mass matrix.

Computing a finite element matrix/vector by *assembly*

Consider computing the mass matrix $M_{jk} = \int_\Omega \phi_j \phi_k \, \mathrm{d}V$, an N by N matrix say, and let's suppose (for clarity only) that we are in 2D.

We **do not** write out the full basis functions explicitly in computing this integral. Instead: firstly, we break the integral down into an integral over elements:

$$M_{jk} = \sum_{\mathcal{K}} \int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$$

Consider $\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$. **Key point**: The only basis functions with are non-zero in the triangle are the 3 basis functions corresponding to the 3 nodes of the element.

Therefore: compute the **elemental contribution to the mass matrix**, a 3 by 3 matrix of the form $\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$ for *3 choices of j and k only.*

Then **add** elemental contribution to full N by N mass matrix.

Computing a finite element matrix/vector by *assembly*

Consider computing the mass matrix $M_{jk} = \int_\Omega \phi_j \phi_k \,\mathrm{d}V$, an N by N matrix say, and let's suppose (for clarity only) that we are in 2D.

We **do not** write out the full basis functions explicitly in computing this integral. Instead: firstly, we break the integral down into an integral over elements:

$$M_{jk} = \sum_{\mathcal{K}} \int_{\mathcal{K}} \phi_j \phi_k \,\mathrm{d}V$$

Consider $\int_{\mathcal{K}} \phi_j \phi_k \,\mathrm{d}V$. **Key point**: The only basis functions with are non-zero in the triangle are the 3 basis functions corresponding to the 3 nodes of the element.

Therefore: compute the **elemental contribution to the mass matrix**, a 3 by 3 matrix of the form $\int_{\mathcal{K}} \phi_j \phi_k \,\mathrm{d}V$ for *3 choices of j and k only*.

Then **add** elemental contribution to full N by N mass matrix.

Computing a finite element matrix/vector by *assembly*

Consider computing the mass matrix $M_{jk} = \int_\Omega \phi_j \phi_k \, \mathrm{d}V$, an N by N matrix say, and let's suppose (for clarity only) that we are in 2D.

We **do not** write out the full basis functions explicitly in computing this integral. Instead: firstly, we break the integral down into an integral over elements:

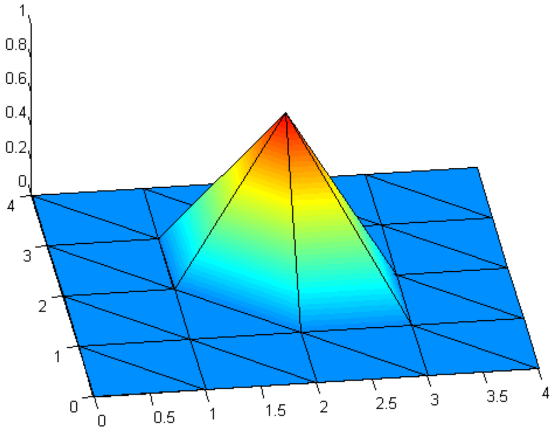
$$M_{jk} = \sum_{\mathcal{K}} \int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$$

Consider $\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$. **Key point**: The only basis functions with are non-zero in the triangle are the 3 basis functions corresponding to the 3 nodes of the element.

Therefore: compute the **elemental contribution to the mass matrix**, a 3 by 3 matrix of the form $\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$ for *3 choices of j and k only*.

Then **add** elemental contribution to full N by N mass matrix.

Assembly

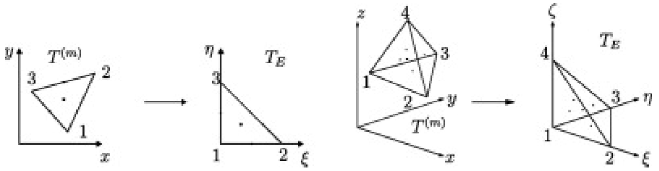


Computing an elemental contribution

We have reduced the problem to computing small matrices/vectors, for example the 3 by 3 matrix

$$\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$$

where ϕ_j, ϕ_k are the 3 basis functions corresponding to the 3 nodes of the mesh.



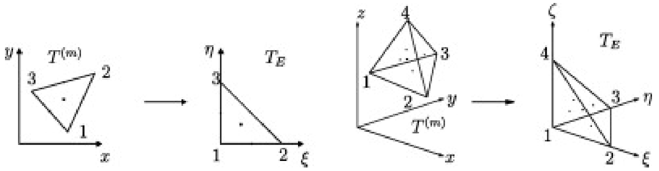
Next, map to the **reference triangle** (also known as the canonical triangle), $\mathcal{K}_{\mathrm{ref}}$, the triangle with nodes $(0,0)$, $(0,1)$, $(1,0)$.

Computing an elemental contribution

We have reduced the problem to computing small matrices/vectors, for example the 3 by 3 matrix

$$\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}V$$

where ϕ_j, ϕ_k are the 3 basis functions corresponding to the 3 nodes of the mesh.



Next, map to the **reference triangle** (also known as the canonical triangle), $\mathcal{K}_{\mathrm{ref}}$, the triangle with nodes $(0,0)$, $(0,1)$, $(1,0)$.

Reference element (also called the canonical element)

We now need to be able to compute

$$\int_{\mathcal{K}} \phi_j \phi_k \, \mathrm{d}x\mathrm{d}y = \int_{\mathcal{K}_{\mathrm{ref}}} \phi_j \phi_k \, \det J \; \mathrm{d}\xi \mathrm{d}\eta$$

where J is the Jacobian of the mapping from the true element to the canonical element.

The basis functions on the reference triangle are easy to write down

$$\begin{aligned} \phi_1(\xi,\eta) &= 1 - \xi - \eta \\ \phi_2(\xi,\eta) &= \xi \\ \phi_3(\xi,\eta) &= \eta \end{aligned}$$

Computing an elemental contribution

J is also required if $\boldsymbol{\nabla}\phi_i$ is needed (for example, in computing the stiffness matrix), since $\boldsymbol{\nabla}\phi_i = J\boldsymbol{\nabla}_{\boldsymbol{\xi}} N_i$.

Consider the mapping from an element with nodes $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, to the canonical element. The inverse mapping can in fact be easily written down using the basis functions.

$$\mathbf{x}(\xi,\eta) = \sum_{j=1}^{3} \mathbf{x}_j N_j(\xi,\eta)$$

from which it is easy to show that J is the following function of nodal positions

$$J = \mathrm{inv}\begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix}$$

Computing an elemental contribution

J is also required if $\nabla\phi_i$ is needed (for example, in computing the stiffness matrix), since $\nabla\phi_i = J\nabla_{\boldsymbol{\xi}} N_i$.

Consider the mapping from an element with nodes $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, to the canonical element. The inverse mapping can in fact be easily written down using the basis functions.

$$\mathbf{x}(\xi,\eta) = \sum_{j=1}^{3} \mathbf{x}_j N_j(\xi,\eta)$$

from which it is easy to show that J is the following function of nodal positions

$$J = \mathrm{inv}\begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix}$$

FEM stages

Solve:

$$\nabla^2 u + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= u^* && \text{on } \Gamma_1 \\ (\nabla u) \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

1. Set up the mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:

$$\begin{aligned} K_{jk} &= \int_\Omega \nabla\phi_j \cdot \nabla\phi_k \, \mathrm{d}V \\ b_j &= \int_\Omega f\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S \end{aligned}$$

3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

This gives the solution $u_h(\mathbf{x}) = U_1\phi_1(\mathbf{x}) + \ldots U_N\phi_N(\mathbf{x})$

FEM stages - full algorithm

Write

$$b_j = \int_\Omega f \phi_j \, \mathrm{d}V + \int_{\Gamma_2} g \phi_j \, \mathrm{d}S$$

as $\mathbf{b} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$

1. Set up the computational mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:
    1. Loop over elements, for each compute the elemental contributions K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ (3 by 3 matrix and 3-vector)
        - For this, need to compute Jacobian J for this element, and loop over quadrature points
    2. Add K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ to K and $\mathbf{b}^{\text{vol}}$ appropriately
    3. Loop over surface-elements on Γ_2, for each compute the elemental contribution $\mathbf{b}^{\text{surf}}_{\text{elem}}$ (a 2-vector).
    4. Add $\mathbf{b}^{\text{surf}}_{\text{elem}}$ to $\mathbf{b}^{\text{surf}}$ appropriately
3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

FEM stages - full algorithm

Write

$$b_j = \int_{\Omega} f\phi_j \,\mathrm{d}V + \int_{\Gamma_2} g\phi_j \,\mathrm{d}S$$

as $\mathbf{b} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$

1. Set up the computational mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:
    1. Loop over elements, for each compute the elemental contributions K_{elem} and $\mathbf{b}^{\text{vol}}_{\mathrm{elem}}$ (3 by 3 matrix and 3-vector)
        - For this, need to compute Jacobian J for this element, and loop over quadrature points
    2. Add K_{elem} and $\mathbf{b}^{\text{vol}}_{\mathrm{elem}}$ to K and $\mathbf{b}^{\text{vol}}$ appropriately
    3. Loop over surface-elements on Γ_2, for each compute the elemental contribution $\mathbf{b}^{\text{surf}}_{\mathrm{elem}}$ (a 2-vector).
    4. Add $\mathbf{b}^{\text{surf}}_{\mathrm{elem}}$ to $\mathbf{b}^{\text{surf}}$ appropriately
3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

FEM stages - full algorithm

Write

$$b_j = \int_\Omega f\phi_j \,\mathrm{d}V + \int_{\Gamma_2} g\phi_j \,\mathrm{d}S$$

as $\mathbf{b} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$

1. Set up the computational mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:
    1. Loop over elements, for each compute the elemental contributions K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ (3 by 3 matrix and 3-vector)
        - For this, need to compute Jacobian J for this element, and loop over quadrature points
    2. Add K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ to K and $\mathbf{b}^{\text{vol}}$ appropriately
    3. Loop over surface-elements on Γ_2, for each compute the elemental contribution $\mathbf{b}^{\text{surf}}_{\text{elem}}$ (a 2-vector).
        - Similar to integrals over elements, again use quadrature
    4. Add $\mathbf{b}^{\text{surf}}_{\text{elem}}$ to $\mathbf{b}^{\text{surf}}$ appropriately
3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

FEM stages - full algorithm

Write

$$b_j = \int_\Omega f \phi_j \,\mathrm{d}V + \int_{\Gamma_2} g \phi_j \,\mathrm{d}S$$

as $\mathbf{b} = \mathbf{b}^{\mathsf{vol}} + \mathbf{b}^{\mathsf{surf}}$

1. Set up the computational mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:
    1. Loop over elements, for each compute the elemental contributions K_{elem} and $\mathbf{b}^{\mathsf{vol}}_{\mathrm{elem}}$ (3 by 3 matrix and 3-vector)
        - For this, need to compute Jacobian J for this element, and loop over quadrature points
    2. Add K_{elem} and $\mathbf{b}^{\mathsf{vol}}_{\mathrm{elem}}$ to K and $\mathbf{b}^{\mathsf{vol}}$ appropriately
    3. Loop over surface-elements on Γ_2, for each compute the elemental contribution $\mathbf{b}^{\mathsf{surf}}_{\mathrm{elem}}$ (a 2-vector).
        - Similar to integrals over elements, again use quadrature
    4. Add $\mathbf{b}^{\mathsf{surf}}_{\mathrm{elem}}$ to $\mathbf{b}^{\mathsf{surf}}$ appropriately
3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

FEM stages - full algorithm

Write

$$b_j = \int_\Omega f \phi_j \, \mathrm{d}V + \int_{\Gamma_2} g \phi_j \, \mathrm{d}S$$

as $\mathbf{b} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$

1. Set up the computational mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:
    1. Loop over elements, for each compute the elemental contributions K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ (3 by 3 matrix and 3-vector)
        - For this, need to compute Jacobian J for this element, and loop over quadrature points
    2. Add K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ to K and $\mathbf{b}^{\text{vol}}$ appropriately
    3. Loop over surface-elements on Γ_2, for each compute the elemental contribution $\mathbf{b}^{\text{surf}}_{\text{elem}}$ (a 2-vector).
        - Similar to integrals over elements, again use quadrature
    4. Add $\mathbf{b}^{\text{surf}}_{\text{elem}}$ to $\mathbf{b}^{\text{surf}}$ appropriately
3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

FEM stages - full algorithm

Write

$$b_j = \int_\Omega f\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S$$

as $\mathbf{b} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$

1. Set up the computational mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:
    1. Loop over elements, for each compute the elemental contributions K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ (3 by 3 matrix and 3-vector)
        - For this, need to compute Jacobian J for this element, and loop over quadrature points
    2. Add K_{elem} and $\mathbf{b}^{\text{vol}}_{\text{elem}}$ to K and $\mathbf{b}^{\text{vol}}$ appropriately
    3. Loop over surface-elements on Γ_2, for each compute the elemental contribution $\mathbf{b}^{\text{surf}}_{\text{elem}}$ (a 2-vector).
        - Similar to integrals over elements, again use quadrature
    4. Add $\mathbf{b}^{\text{surf}}_{\text{elem}}$ to $\mathbf{b}^{\text{surf}}$ appropriately
3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

FEM stages - full algorithm

Write

$$b_j = \int_\Omega f\phi_j \,\mathrm{d}V + \int_{\Gamma_2} g\phi_j \,\mathrm{d}S$$

as $\mathbf{b} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$

1. Set up the computational mesh and choose basis functions
2. Compute the matrix K and vector $\mathbf{b}$:
    1. Loop over elements, for each compute the elemental contributions K_{elem} and $\mathbf{b}^{\text{vol}}_{\mathrm{elem}}$ (3 by 3 matrix and 3-vector)
        - For this, need to compute Jacobian J for this element, and loop over quadrature points
    2. Add K_{elem} and $\mathbf{b}^{\text{vol}}_{\mathrm{elem}}$ to K and $\mathbf{b}^{\text{vol}}$ appropriately
    3. Loop over surface-elements on Γ_2, for each compute the elemental contribution $\mathbf{b}^{\text{surf}}_{\mathrm{elem}}$ (a 2-vector).
        - Similar to integrals over elements, again use quadrature
    4. Add $\mathbf{b}^{\text{surf}}_{\mathrm{elem}}$ to $\mathbf{b}^{\text{surf}}$ appropriately
3. Alter linear system $K\mathbf{U} = \mathbf{b}$ to impose Dirichlet BCs
4. Solve linear system

FEM for simple PDEs: Object-oriented implementation (general ideas)

Note that in the following:

- We consider *one possible* approach - the appropriate design will depend fundamentally on the precise nature of the solver required (eg, a solver for a particular equation versus a general solver of several)
- `Purple` represents an abstract class/method, `red` represents a concrete class or implemented method, `blue` represents a self-contained class (no inheritance).
- Important members or methods of the classes will be given, but obvious extra methods will be omitted, such as Get/Set methods

FEM for simple PDEs: Object-oriented implementation (general ideas)

Note that in the following:

- We consider *one possible* approach - the appropriate design will depend fundamentally on the precise nature of the solver required (eg, a solver for a particular equation versus a general solver of several)
- `Purple` represents an abstract class/method, `red` represents a concrete class or implemented method, `blue` represents a self-contained class (no inheritance).
- Important members or methods of the classes will be given, but obvious extra methods will be omitted, such as Get/Set methods

FEM for simple PDEs: Object-oriented implementation (general ideas)

Note that in the following:

- We consider *one possible* approach - the appropriate design will depend fundamentally on the precise nature of the solver required (eg, a solver for a particular equation versus a general solver of several)
- `Purple` represents an abstract class/method, `red` represents a concrete class or implemented method, `blue` represents a self-contained class (no inheritance).
- Important members or methods of the classes will be given, but obvious extra methods will be omitted, such as Get/Set methods

Object-oriented design

What are the self-contained ‘concepts’ (objects) that form the overall simulation code, and what functionality should each of these objects have?

Geometry

`Node`

Member var: `mLocation`

- ▷ *a vector*

Member var: `mIsBoundaryNode`

- ▷ *a boolean (true/false)*

`Element`

Member var: `mNodes`

- ▷ *(Pointers to) the 3 nodes (assuming a 2d simulation) of this element*

Method: `ComputeJacobian()`

Method: `ComputeJacobianDeterminant()`

`SurfaceElement`

Member var: `mNodes`

- ▷ *(Pointers to) the 2 nodes of this element*
- ▷ *Also has corresponding methods to the Jacobian methods above*

Geometry

Mesh

- `mNodes`
  - ▷ *a list of `Node` objects*
- `mElements`
  - ▷ *a list of `Element` objects*
- `mBoundaryElements`
  - ▷ *a list of surface elements (`SurfaceElement`) on the boundary*
- *Method:* `ReadFromFile(filename)`
- *Method:* `GenerateRegularMesh(width,height,stepsize)`
- *Method:* `Refine()`

Note

- Here, boundary nodes/elements represent the *entire* boundary—'mesh' concept is self-contained and **not** dependent on PDE problem being solved.

Boundary conditions

- There are various ways this could be implemented
- **Key point**: the implementation requires that
  - Dirichlet BCs be defined at boundary *nodes*
  - Neumann BCs be defined on boundary *elements* (ie element interiors)

```
BoundaryConditions<DIM>

    mDirichletBoundaryNodes
    mDirichletValues
    mNeumannBoundaryElements
    mNeumannValues

    AddDirichletBoundaryCondition(node,dirichletBcValue)
    AddNeumannBoundaryCondition(boundaryElement,neumannBcValue)
```

A simple solver

Suppose we want to write a solver for Poisson's equation $\nabla^2 u = f$ for general forcing terms $f(\mathbf{x})$ and general boundary conditions. The solver class could be self-contained, and look like:

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \,\mathrm{d}V$
    - Loop over elements of mesh (“`mesh.GetNumElements()`”, “`mesh.GetElement(i)`”)
    - For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    - Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\mathsf{vol}} = \int_\Omega f \phi_i \,\mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\mathsf{surf}} = \int_{\Gamma_2} g \phi_i \,\mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \, \mathrm{d}V$
   - Loop over elements of mesh ("`mesh.GetNumElements()`", "`mesh.GetElement(i)`")
   - For each element set-up the elemental stiffness matrix - loop over quadrature points, call `element.GetJacobian()` etc
   - Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\text{vol}} = \int_\Omega f \phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\text{surf}} = \int_{\Gamma_2} g \phi_i \, \mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \,\mathrm{d}V$
    1. Loop over elements of mesh (“`mesh.GetNumElements()`”, “`mesh.GetElement(i)`”)
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\text{vol}} = \int_\Omega f\phi_i \,\mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\text{surf}} = \int_{\Gamma_2} g\phi_i \,\mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \, \mathrm{d}V$
    1. Loop over elements of mesh (“`mesh.GetNumElements()`”, “`mesh.GetElement(i)`”)
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\text{vol}} = \int_\Omega f\phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\text{surf}} = \int_{\Gamma_2} g\phi_i \, \mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \,\mathrm{d}V$
    1. Loop over elements of mesh ("`mesh.GetNumElements()`", "`mesh.GetElement(i)`")
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\text{vol}} = \int_\Omega f \phi_i \,\mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\text{surf}} = \int_{\Gamma_2} g \phi_i \,\mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \,\mathrm{d}V$
    1. Loop over elements of mesh ("`mesh.GetNumElements()`", "`mesh.GetElement(i)`")
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\text{vol}} = \int_\Omega f\phi_i \,\mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\text{surf}} = \int_{\Gamma_2} g\phi_i \,\mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \,\mathrm{d}V$
    1. Loop over elements of mesh ("`mesh.GetNumElements()`", "`mesh.GetElement(i)`")
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\mathbf{vol}} = \int_\Omega f \phi_i \,\mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\mathbf{surf}} = \int_{\Gamma_2} g \phi_i \,\mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \, \mathrm{d}V$
    1. Loop over elements of mesh (“`mesh.GetNumElements()`”, “`mesh.GetElement(i)`”)
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\mathbf{vol}} = \int_\Omega f \phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\mathbf{surf}} = \int_{\Gamma_2} g \phi_i \, \mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \, \mathrm{d}V$
    1. Loop over elements of mesh ("`mesh.GetNumElements()`", "`mesh.GetElement(i)`")
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\mathbf{vol}} = \int_\Omega f \phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\mathbf{surf}} = \int_{\Gamma_2} g \phi_i \, \mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

```
PoissonEquationSolver:
    Solve(mesh,abstractForce,boundaryConditions)
```

The `Solve` method needs to:

1. Initialise a matrix K and vector $\mathbf{b}$
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \, \mathrm{d}V$
    1. Loop over elements of mesh ("`mesh.GetNumElements()`", "`mesh.GetElement(i)`")
    2. For each element set-up the elemental stiffness matrix – loop over quadrature points, call `element.GetJacobian()` etc
    3. Add elemental contribution to K
3. Similarly, loop over elements and assemble $b_i^{\mathbf{vol}} = \int_\Omega f \phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\mathbf{surf}} = \int_{\Gamma_2} g \phi_i \, \mathrm{d}S$
5. Alter the linear system $K\mathbf{U} = \mathbf{b}^{\mathrm{vol}} + \mathbf{b}^{\mathrm{surf}}$ to take the Dirichlet BCs into account (using `boundaryConditions` again).
6. Solve the linear system

A simple solver

A (possible) more complete interface

```
PoissonEquationSolver:
    PoissonEquationSolver(mesh,abstractForce,boundaryConditions)
    Solve()
    PlotSolution()
    PlotForce()
    ComputeQOI()
    WriteSolutionToFile(filename)
```