FEM for simple PDEs: Object-oriented implementation (introduction)

Template classes (C++)

C++ allows you to do:

```
template<int DIM>
class Node
{
    // use DIM in some way
}
```

from which the compiler creates different versions of the class, depending on which values of DIM is used. This is an alternative to having a member variable `mDimension` inside the class.

Usage:

```
Node<3> 3d_node;
Node<2> 2d_node;
```

This kind of code would generally a compile error (which is good):

```
Node<3> node;
Mesh<2> mesh;
mesh.AddNode(node);
```

Template classes (C++)

C++ allows you to do:

```
template<int DIM>
class Node
{
    // use DIM in some way
}
```

from which the compiler creates different versions of the class, depending on which values of DIM is used. This is an alternative to having a member variable `mDimension` inside the class.

Usage:

```
Node<3> 3d_node;
Node<2> 2d_node;
```

This kind of code would generally a compile error (which is good):

```
Node<3> node;
Mesh<2> mesh;
mesh.AddNode(node);
```

Template classes (C++)

C++ allows you to do:

```
template<int DIM>
class Node
{
    // use DIM in some way
}
```

from which the compiler creates different versions of the class, depending on which values of DIM is used. This is an alternative to having a member variable `mDimension` inside the class.

Usage:

```
Node<3> 3d_node;
Node<2> 2d_node;
```

This kind of code would generally a compile error (which is good):

```
Node<3> node;
Mesh<2> mesh;
mesh.AddNode(node);
```

The procedural approach

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)
- Related to Chaste design but heavily simplified
- `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)
- Related to Chaste design but heavily simplified
- `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)
- Related to Chaste design but heavily simplified
- `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*

`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 - using templates

`Node<SPACE_DIM>`

Member var: `mLocation`

- ▷ *a vector of length* `SPACE_DIM`

`Element<ELEM_DIM,SPACE_DIM>`

Member var: `mNodes`

- ▷ *(Pointers to) the nodes of this element*

Method: `ComputeJacobian()` etc, depending on dimensions

Then:

- `Element<2,2>` represents a volume element
- `Element<1,2>` represents a surface element

Geometry - using templates

`Node<SPACE_DIM>`

Member var: `mLocation`

▷ *a vector of length* `SPACE_DIM`

`Element<ELEM_DIM,SPACE_DIM>`

Member var: `mNodes`

▷ *(Pointers to) the nodes of this element*

Method: `ComputeJacobian()` etc, depending on dimensions

Then:

- `Element<2,2>` represents a volume element
- `Element<1,2>` represents a surface element

Geometry

`Mesh<DIM>`

- `mNodes`
  - ▷ *a list of `Node<DIM>` objects*
- `mElements`
  - ▷ *a list of `Element<DIM,DIM>` objects*
- `mBoundaryElements`
  - ▷ *a list of surface elements (`Element<DIM-1,DIM>`) on the boundary*
- `mBoundaryNodeIndices`

Note:

- There are other possibilities (nodes knowing whether they are a boundary node, for example)
- Here, boundary nodes/elements represent the *entire* boundary—'mesh' concept is self-contained and not dependent on PDE problem being solved.

Basis functions

If solving a problem with piece-wise linear basis functions:

`LinearBasisFunction<ELEM_DIM>`

`GetValues(xi)`

- ▷ *xi is a vector of size ELEM_DIM, and this function returns the vector* $[N_1(\boldsymbol{\xi}), \ldots, N_n(\boldsymbol{\xi})] = [\phi_1(\mathbf{x}(\boldsymbol{\xi})), \ldots, \phi_n(\mathbf{x}(\boldsymbol{\xi}))]$

`GetTransformedDerivatives(xi, J)`

- ▷ *similarly, returns vector with entries* $\boldsymbol{\nabla}\phi_i = J\boldsymbol{\nabla}_{\boldsymbol{\xi}} N_i$

There are again other possibilities, eg. just having `GetDerivatives(xi)` and having calling code deal with multiplication by J, or doing:

`AbstractBasisFunction<ELEM_DIM>`:

`GetValues(xi)`

`GetTransformedDerivatives(xi, J)`

and then having `LinearBasisFunction` and `QuadraticBasisFunction`

Basis functions

If solving a problem with piece-wise linear basis functions:

`LinearBasisFunction<ELEM_DIM>`

- `GetValues(xi)`
  - ▷ *xi is a vector of size ELEM_DIM, and this function returns the vector* $[N_1(\boldsymbol{\xi}), \ldots, N_n(\boldsymbol{\xi})] = [\phi_1(\mathbf{x}(\boldsymbol{\xi})), \ldots, \phi_n(\mathbf{x}(\boldsymbol{\xi}))]$
- `GetTransformedDerivatives(xi, J)`
  - ▷ *similarly, returns vector with entries* $\boldsymbol{\nabla}\phi_i = J\boldsymbol{\nabla}_{\boldsymbol{\xi}} N_i$

There are again other possibilities, eg. just having `GetDerivatives(xi)` and having calling code deal with multiplication by J, or doing:

`AbstractBasisFunction<ELEM_DIM>`:

- `GetValues(xi)`
- `GetTransformedDerivatives(xi, J)`

and then having `LinearBasisFunction` and `QuadraticBasisFunction`

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. Set up a `LinearBasisFunction` object
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \, \mathrm{d}V$
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. Set up a `LinearBasisFunction` object
2. Set up stiffness matrix $K_{ij} = \int_\Omega \phi_i \phi_j \, \mathrm{d}V$
   a. Loop over elements of mesh (`"mesh.GetNumElements()"`, `"mesh.GetElement(i)"`)
   b. For each element, set up the element stiffness matrix using quadrature points, call `element.GetTransformation()`, `basis.Eval`, `GetIntegrationPoint` etc.
   c. Add elemental contributions 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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. Set up a `LinearBasisFunction` object
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()` and `basis_func.GetValues(xi)` 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

FEM for simple PDEs: Object-oriented implementation in Chaste

Mesh classes in Chaste

`Node<SPACE_DIM>`

- ▷ *data includes: location, index, whether it is a boundary node*

`AbstractElement<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains nodes, not necessarily tetrahedral*

`AbstractTetrahedralElement<ELEM_DIM,SPACE_DIM>`

- ▷ *Methods to calculate the jacobian, etc*

`Element<ELEM_DIM,SPACE_DIM>`

`AbstractMesh<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains nodes but not elements*

`AbstractTetrahedralMesh<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains elements, access methods, and lots of functionality*

`TetrahedralMesh<ELEM_DIM,SPACE_DIM>` and `DistributedTetrahedralMesh<ELEM_DIM,SPACE_DIM>`

There are also `MutableMesh`, `Cylindrical2dMesh` (both for cell-based simulations), `QuadraticMesh`, and more..

Mesh classes in Chaste

`Node<SPACE_DIM>`

- ▷ *data includes: location, index, whether it is a boundary node*

`AbstractElement<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains nodes, not necessarily tetrahedral*

`AbstractTetrahedralElement<ELEM_DIM,SPACE_DIM>`

- ▷ *Methods to calculate the jacobian, etc*

`Element<ELEM_DIM,SPACE_DIM>`

`AbstractMesh<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains nodes but not elements*

`AbstractTetrahedralMesh<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains elements, access methods, and lots of functionality*

`TetrahedralMesh<ELEM_DIM,SPACE_DIM>` and `DistributedTetrahedralMesh<ELEM_DIM,SPACE_DIM>`

There are also `MutableMesh`, `Cylindrical2dMesh` (both for cell-based simulations), `QuadraticMesh`, and more..

Mesh classes in Chaste

`Node<SPACE_DIM>`

- ▷ *data includes: location, index, whether it is a boundary node*

`AbstractElement<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains nodes, not necessarily tetrahedral*

`AbstractTetrahedralElement<ELEM_DIM,SPACE_DIM>`

- ▷ *Methods to calculate the jacobian, etc*

`Element<ELEM_DIM,SPACE_DIM>`

`AbstractMesh<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains nodes but not elements*

`AbstractTetrahedralMesh<ELEM_DIM,SPACE_DIM>`

- ▷ *Contains elements, access methods, and lots of functionality*

`TetrahedralMesh<ELEM_DIM,SPACE_DIM>` and `DistributedTetrahedralMesh<ELEM_DIM,SPACE_DIM>`

There are also `MutableMesh`, `Cylindrical2dMesh` (both for cell-based simulations), `QuadraticMesh`, and more..

Basis functions and BCC in Chaste

`LinearBasisFunction` defined as above, (just static methods), and similarly, `QuadraticBasisFunction` (no inheritance).

`BoundaryConditionsContainer`

- ▷ *Same as the '`BoundaryConditions`' class outlined above.*
- ▷ *Contains Dirichlet nodes and corresponding BC values*
- ▷ *Contains Neumann boundary elements and corresponding BC*
- ▷ *Method for applying the Dirichlet BCs to a supplied linear system*

Some discretisations Chaste is required to solve

Consider the discretised heat equation

$$(M + \Delta t\, K)\, \mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, \mathbf{b}^{\mathrm{vol},n} + \Delta t\, \mathbf{b}^{\mathrm{surf},n}$$

which requires M, K, $\mathbf{b}^{\mathrm{vol},n}$ and $\mathbf{b}^{\mathrm{surf},n}$ to be 'assembled'

The following is a discretisation that arises in cardiac electro-physiology

$$(M + \Delta t\, K)\, \mathbf{V}^{n+1} = M\mathbf{V}^n + \Delta t\, M\mathbf{F}^n + \Delta t\, \mathbf{c}^n + \Delta t\, \mathbf{b}^{\mathrm{surf},n}$$

where

- $\mathbf{F}^n$ represents nodal ionic currents
- $\mathbf{c}^n$ is a correction term that improves accuracy
- $\mathbf{b}^{\mathrm{surf},n}$ is an integral over a 1D cable structure

Some discretisations Chaste is required to solve

Consider the discretised heat equation

$$(M + \Delta t\, K)\, \mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, \mathbf{b}^{\mathrm{vol},n} + \Delta t\, \mathbf{b}^{\mathrm{surf},n}$$

which requires M, K, $\mathbf{b}^{\mathrm{vol},n}$ and $\mathbf{b}^{\mathrm{surf},n}$ to be 'assembled'

The following is a discretisation that arises in cardiac electro-physiology

$$(M + \Delta t\, K)\, \mathbf{V}^{n+1} = M\mathbf{V}^n + \Delta t\, M\mathbf{F}^n + \Delta t\, \mathbf{c}^n + \Delta t\, \mathbf{d}^n_{\mathrm{purkinje}}$$

where

- $\mathbf{F}^n$ represents nodal ionic currents
- $\mathbf{c}^n$ is a correction term that improves accuracy
- $\mathbf{d}^n_{\mathrm{purkinje}}$ is an integral over a 1D-sub-structure

Some discretisations Chaste is required to solve

Consider the discretised heat equation

$$(M + \Delta t\, K)\, \mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, \mathbf{b}^{\mathrm{vol},n} + \Delta t\, \mathbf{b}^{\mathrm{surf},n}$$

which requires M, K, $\mathbf{b}^{\mathrm{vol},n}$ and $\mathbf{b}^{\mathrm{surf},n}$ to be 'assembled'

The following is a discretisation that arises in cardiac electro-physiology

$$(M + \Delta t\, K)\, \mathbf{V}^{n+1} = M\mathbf{V}^n + \Delta t\, M\mathbf{F}^n + \Delta t\, \mathbf{c}^n + \Delta t\, \mathbf{d}^n_{\mathrm{purkinje}}$$

where

- $\mathbf{F}^n$ represents nodal ionic currents
- $\mathbf{c}^n$ is a correction term that improves accuracy
- $\mathbf{d}^n_{\mathrm{purkinje}}$ is an integral over a 1D-sub-structure

Some discretisations Chaste is required to solve

Consider the discretised heat equation

$$(M + \Delta t\, K)\, \mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, \mathbf{b}^{\mathrm{vol},n} + \Delta t\, \mathbf{b}^{\mathrm{surf},n}$$

which requires M, K, $\mathbf{b}^{\mathrm{vol},n}$ and $\mathbf{b}^{\mathrm{surf},n}$ to be 'assembled'

The following is a discretisation that arises in cardiac electro-physiology

$$(M + \Delta t\, K)\, \mathbf{V}^{n+1} = M\mathbf{V}^n + \Delta t\, M\mathbf{F}^n + \Delta t\, \mathbf{c}^n + \Delta t\, \mathbf{d}^n_{\mathrm{purkinje}}$$

where

- $\mathbf{F}^n$ represents nodal ionic currents
- $\mathbf{c}^n$ is a correction term that improves accuracy
- $\mathbf{d}^n_{\mathrm{purkinje}}$ is an integral over a 1D-sub-structure

Solvers versus assemblers

The requirements of Chaste to solve a variety of problem (and using various discretisations) suggest the following type of design:

Assembler classes

- used to construct any 'finite element' matrix or vector, i.e. something that requires a loop over elements (or surface-elements) etc, to be set up, such as M, K etc.

Solver classes

- these **use** assemblers to set up a particular linear system, then solve it

Solvers versus assemblers

The requirements of Chaste to solve a variety of problem (and using various discretisations) suggest the following type of design:

Assembler classes

- used to construct any 'finite element' matrix or vector, i.e. something that requires a loop over elements (or surface-elements) etc, to be set up, such as M, K etc.

Solver classes

- these **use** assemblers to set up a particular linear system, then solve it

Solvers versus assemblers

The requirements of Chaste to solve a variety of problem (and using various discretisations) suggest the following type of design:

Assembler classes

- used to construct any 'finite element' matrix or vector, i.e. something that requires a loop over elements (or surface-elements) etc, to be set up, such as M, K etc.

Solver classes

- these **use** assemblers to set up a particular linear system, then solve it

Assembler concept

Consider computing any of the following

$$
\begin{aligned}
M_{jk} &= \int_\Omega \phi_j \phi_k \, \mathrm{d}V \\
K_{jk} &= \int_\Omega \boldsymbol{\nabla} \phi_j \cdot D \boldsymbol{\nabla} \phi_k \, \mathrm{d}V \\
b_j^{\mathrm{vol}} &= \int_\Omega f \phi_j \, \mathrm{d}V
\end{aligned}
$$

1. Loop over elements, for each compute the elemental contributions K_{elem} or M_{elem} or $\mathbf{b}_{\mathrm{elem}}^{\mathrm{vol}}$ (3 by 3 matrices or 3-vector)
   - For this, need to compute Jacobian J for this element, and loop over quadrature points
2. Add K_{elem} or M_{elem} or $\mathbf{b}_{\mathrm{elem}}^{\mathrm{vol}}$ to full matrix appropriately

Assembler concept

Consider computing any of the following

$$
\begin{aligned}
M_{jk} &= \int_\Omega \phi_j \phi_k \, \mathrm{d}V \\
K_{jk} &= \int_\Omega \boldsymbol{\nabla} \phi_j \cdot D \boldsymbol{\nabla} \phi_k \, \mathrm{d}V \\
b_j^{\mathrm{vol}} &= \int_\Omega f \phi_j \, \mathrm{d}V
\end{aligned}
$$

1. Loop over elements, for each compute the elemental contributions K_{elem} or M_{elem} or $\mathbf{b}_{\mathrm{elem}}^{\mathrm{vol}}$ (3 by 3 matrices or 3-vector)
   - For this, need to compute Jacobian J for this element, and loop over quadrature points
2. Add K_{elem} or M_{elem} or $\mathbf{b}_{\mathrm{elem}}^{\mathrm{vol}}$ to full matrix appropriately

Assembler concept

In all cases we can write the integral over the element as

$$\int_{\mathcal{K}_{\text{ref}}} \mathcal{F}(x, y, u, \phi_1, \phi_2, \phi_3, \boldsymbol{\nabla}\phi_1, \boldsymbol{\nabla}\phi_2, \boldsymbol{\nabla}\phi_3) \det J \, \mathrm{d}\xi \mathrm{d}\eta$$

where

Computing mass matrix	$\Rightarrow$	$\mathcal{F}$ is the matrix $\phi_j \phi_k$
Computing stiffness matrix	$\Rightarrow$	$\mathcal{F}$ is the matrix $\boldsymbol{\nabla}\phi_j \cdot D\boldsymbol{\nabla}\phi_k$
Computing $\mathbf{b}^{\text{vol}}$	$\Rightarrow$	$\mathcal{F}$ is the vector $f\phi_j$

`AbstractAssembler`

▷ *Does everything above except provide the form of* $\mathcal{F}$

Abs. method: A method representing $\mathcal{F}$

`MassMatrixAssembler` *inherits from* `AbstractAssembler`:

Implemented method: $\mathcal{F}$ returns the matrix $\phi_j \phi_k$

Assembler classes: abstract base class

Define an (essentially) abstract class `AbstractFeObjectAssembler`, which is templated over the dimensions, and also booleans saying whether the class will assemble matrices (eg M, K) and/or vectors (eg $\mathbf{b}^{\text{vol}}$).

`AbstractFeObjectAssembler<DIMs,CAN_ASSEMBLE_VEC,CAN_ASSEMBLE_MAT>`

- `SetMatrixToBeAssembled(matrix)`
- `SetVectorToBeAssembled(vector)`
- `Assemble()`
  - ▷ *Loops over elements, computes elemental contribution by calling:*
- `AssembleOnElement(..)`
  - ▷ *Computes element contribution by looping over quadrature points, and at each quad point calling one or both of the following:*
- `ComputeMatrixTerm(..)`
  - ▷ *the function $\mathcal{F}$ for matrices*
- `ComputeVectorTerm(..)`
  - ▷ *the function $\mathcal{F}$ for vectors*

Assembler classes: example concrete classes

`MassMatrixAssembler` *inherits from* `AbsFeObjectAssembler<false,true>` :

Implemented method: `ComputeMatrixTerm(..)`

▷ *return matrix* $\phi_j \phi_k$ *(elemental-contribution, 3 by 3 matrix in 2D)*

`StiffnessMatrixAssembler` *inherits from* `AbsFeObjectAssembler<false, true>`:

Implemented method: `ComputeMatrixTerm(..)`

▷ *return matrix* $\nabla \phi_j \cdot \nabla \phi_k$ *(elemental-contribution)*

This designs allows new assemblers to be written fairly easily, and provides the flexibility required of the code

Assembler classes: example concrete classes

`MassMatrixAssembler` *inherits from* `AbsFeObjectAssembler<false,true>` :

Implemented method: `ComputeMatrixTerm(..)`

▷ *return matrix* $\phi_j \phi_k$ *(elemental-contribution, 3 by 3 matrix in 2D)*

`StiffnessMatrixAssembler` *inherits from* `AbsFeObjectAssembler<false, true>`:

Implemented method: `ComputeMatrixTerm(..)`

▷ *return matrix* $\nabla \phi_j \cdot \nabla \phi_k$ *(elemental-contribution)*

This designs allows new assemblers to be written fairly easily, and provides the flexibility required of the code

Assembler classes: example concrete classes

`MassMatrixAssembler` *inherits from* `AbsFeObjectAssembler<false,true>` :

Implemented method: `ComputeMatrixTerm(..)`

▷ *return matrix* $\phi_j \phi_k$ *(elemental-contribution, 3 by 3 matrix in 2D)*

`StiffnessMatrixAssembler` *inherits from* `AbsFeObjectAssembler<false, true>`:

Implemented method: `ComputeMatrixTerm(..)`

▷ *return matrix* $\nabla \phi_j \cdot \nabla \phi_k$ *(elemental-contribution)*

This designs allows new assemblers to be written fairly easily, and provides the flexibility required of the code

Solver classes in Chaste

`AbstractLinearPdeSolver`:

`SetupLinearSystem()`

- ▷ *Needs to be implemented in concrete class, and should fully set up the linear system for the particular problem being solved*

`AbstractStaticPdeSolver` *inherits from* `AbstractLinearPdeSolver`:

`Solve()`

- ▷ *Calls* `SetupLinearSystem()` *and then solves linear system*

`AbstractDynamicPdeSolver` *inherits from* `AbstractLinearPdeSolver`:

`SetTimes(t0,t1)`

`SetInitialCondition(initialCondition)`

`Solve()`

- ▷ *Repeatedly calls* `SetupLinearSystem()` *and solves linear system*

Solver classes in Chaste

`AbstractLinearPdeSolver`:

`SetupLinearSystem()`

- ▷ *Needs to be implemented in concrete class, and should fully set up the linear system for the particular problem being solved*

`AbstractStaticPdeSolver` *inherits from* `AbstractLinearPdeSolver`:

`Solve()`

- ▷ *Calls* `SetupLinearSystem()` *and then solves linear system*

`AbstractDynamicPdeSolver` *inherits from* `AbstractLinearPdeSolver`:

`SetTimes(t0,t1)`

`SetInitialCondition(initialCondition)`

`Solve()`

- ▷ *Repeatedly calls* `SetupLinearSystem()` *and solves linear system*

Solver classes in Chaste

`AbstractLinearPdeSolver`:

`SetupLinearSystem()`

- ▷ *Needs to be implemented in concrete class, and should fully set up the linear system for the particular problem being solved*

`AbstractStaticPdeSolver` *inherits from* `AbstractLinearPdeSolver`:

`Solve()`

- ▷ *Calls* `SetupLinearSystem()` *and then solves linear system*

`AbstractDynamicPdeSolver` *inherits from* `AbstractLinearPdeSolver`:

```
SetTimes(t0,t1)
SetInitialCondition(initialCondition)
Solve()
```

- ▷ *Repeatedly calls* `SetupLinearSystem()` *and solves linear system*

Example usage of the general design

The discretisation for the monodomain equation (cardiac electro-physiology)

$$(M + \Delta t\, K)\, \mathbf{V}^{n+1} = M\mathbf{V}^n + \Delta t\, M\mathbf{F}^n + \Delta t\, \mathbf{c}^n$$

where only the highlighted terms are 'assembled'.

Write concrete classes

- `MassMatrixAssembler` for computing M
- `MonodomainAssembler` for computing $M + \Delta t\, K$
- `CorrectionTermAssembler` for computing $\mathbf{c}^n$

`MonodomainSolver` *inherits from* `AbstractDynamicPdeSolver`:

Member var: `mMassMatrixAssembler`
Member var: `mMonodomainAssembler`
Member var: `mCorrectionTermAssembler`
Implemented method: `SetUpLinearSystem()`

▷ *Uses the above assemblers to set up the linear system*

Example usage of the general design

The discretisation for the monodomain equation (cardiac electro-physiology)

$$(M + \Delta t\, K)\, \mathbf{V}^{n+1} = M\mathbf{V}^n + \Delta t\, M\mathbf{F}^n + \Delta t\, \mathbf{c}^n$$

where only the highlighted terms are 'assembled'.

Write concrete classes

- `MassMatrixAssembler` for computing M
- `MonodomainAssembler` for computing $M + \Delta t\, K$
- `CorrectionTermAssembler` for computing $\mathbf{c}^n$

`MonodomainSolver` *inherits from* `AbstractDynamicPdeSolver`:

Member var: `mMassMatrixAssembler`

Member var: `mMonodomainAssembler`

Member var: `mCorrectionTermAssembler`

Implemented method: `SetUpLinearSystem()`

▷ *Uses the above assemblers to set up the linear system*

Example usage of the general design

The discretisation for the monodomain equation (cardiac electro-physiology)

$$(M + \Delta t\, K)\,\mathbf{V}^{n+1} = M\mathbf{V}^{n} + \Delta t\, M\mathbf{F}^{n} + \Delta t\, \mathbf{c}^{n}$$

where only the highlighted terms are 'assembled'.

Write concrete classes

- `MassMatrixAssembler` for computing M
- `MonodomainAssembler` for computing $M + \Delta t\, K$
- `CorrectionTermAssembler` for computing $\mathbf{c}^{n}$

`MonodomainSolver` *inherits from* `AbstractDynamicPdeSolver`:

Member var: `mMassMatrixAssembler`
Member var: `mMonodomainAssembler`
Member var: `mCorrectionTermAssembler`
Implemented method: `SetUpLinearSystem()`

▷ *Uses the above assemblers to set up the linear system*

Example usage of the general design

An alternative discretisation (Crank-Nicolson, i.e. the trapezoidal rule)

$$\left(M + \frac{1}{2}\Delta t\, K\right)\mathbf{V}^{n+1} = \left(M - \frac{1}{2}\Delta t\, K\right)\mathbf{V}^{n} + \Delta t\, M\mathbf{F}^{n} + \Delta t\, \mathbf{c}^{n}$$

where the highlighted terms are 'assembled'.

`CrankNicolsonMonodomainSolver`[1] *inherits from* `AbsDynamicPdeSolver` :

- *Member var:* `mMassMatrixAssembler`
- *Member var:* `mStiffnessMatrixAssembler`
- *Member var:* `mCorrectionTermAssembler`
- *Implemented method:* `SetUpLinearSystem()`
  - ▷ **Uses** *the above assemblers to set up this linear system*

[1] This class doesn't exist (yet), the point is that the design allows it to be implemented fairly easily

Example usage of the general design

An alternative discretisation (Crank-Nicolson, i.e. the trapezoidal rule)

$$\left(M + \frac{1}{2}\Delta t\, K\right)\mathbf{V}^{n+1} = \left(M - \frac{1}{2}\Delta t\, K\right)\mathbf{V}^{n} + \Delta t\, M\mathbf{F}^{n} + \Delta t\, \mathbf{c}^{n}$$

where the highlighted terms are 'assembled'.

`CrankNicolsonMonodomainSolver`[1] *inherits from* `AbsDynamicPdeSolver` :

Member var: `mMassMatrixAssembler`

Member var: `mStiffnessMatrixAssembler`

Member var: `mCorrectionTermAssembler`

Implemented method: `SetUpLinearSystem()`

▷ **Uses** *the above assemblers to set up this linear system*

[1] This class doesn't exist (yet), the point is that the design allows it to be implemented fairly easily

Solver-assembler objects

For some problems and with simple discretisations the linear system is of the form $A\mathbf{U}^n = \mathbf{B}$, where both A and $\mathbf{B}$ are 'assembled'.

For example, for the general elliptic problem $\nabla \cdot (D\nabla u) + f = 0$ (with BCs), the discretisation is $K\mathbf{U} = \mathbf{b}$ as we have seen

Also, for the parabolic problem $u_t = \nabla \cdot (D\nabla u) + f$ (with BCs), the discretisation can be written as

$$A\mathbf{U}^{n+1} = \mathbf{B}$$

where

$$A_{jk} = \int_\Omega \phi_j \phi_k + \Delta t \, \nabla \phi_j \cdot \nabla \phi_k \, \mathrm{d}V$$

$$B_j = \int_\Omega (u^n + f)\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S$$

Solver-assembler objects

For some problems and with simple discretisations the linear system is of the form $A\mathbf{U}^n = \mathbf{B}$, where both A and $\mathbf{B}$ are 'assembled'.

For example, for the general elliptic problem $\nabla \cdot (D\nabla u) + f = 0$ (with BCs), the discretisation is $K\mathbf{U} = \mathbf{b}$ as we have seen

Also, for the parabolic problem $u_t = \nabla \cdot (D\nabla u) + f$ (with BCs), the discretisation can be written as

$$A\mathbf{U}^{n+1} = \mathbf{B}$$

where

$$A_{jk} = \int_\Omega \phi_j \phi_k + \Delta t \, \nabla \phi_j \cdot \nabla \phi_k \, \mathrm{d}V$$

$$B_j = \int_\Omega (u^n + f)\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S$$

Solver-assembler objects

For some problems and with simple discretisations the linear system is of the form $A\mathbf{U}^n = \mathbf{B}$, where both A and $\mathbf{B}$ are 'assembled'.

For example, for the general elliptic problem $\nabla \cdot (D\boldsymbol{\nabla} u) + f = 0$ (with BCs), the discretisation is $K\mathbf{U} = \mathbf{b}$ as we have seen

Also, for the parabolic problem $u_t = \nabla \cdot (D\boldsymbol{\nabla} u) + f$ (with BCs), the discretisation can be written as

$$A\mathbf{U}^{n+1} = \mathbf{B}$$

where

$$A_{jk} = \int_\Omega \phi_j \phi_k + \Delta t \, \boldsymbol{\nabla}\phi_j \cdot \boldsymbol{\nabla}\phi_k \, \mathrm{d}V$$

$$B_j = \int_\Omega (u^n + f)\phi_j \, \mathrm{d}V + \int_{\Gamma_2} g\phi_j \, \mathrm{d}S$$

Solver-assembler objects

The original Chaste design just considered such problems, and for these problems solvers don't need to own assemblers—**solvers are assemblers**. The concrete 'assembler-solver' class for a particular problem needs to implement `ComputeMatrixTerm()`, `ComputeVectorTerm()` etc. This design pattern is still used:

`SimpleLinearEllipticSolver` essentially inherits from *both* `AbstractStaticPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

`SimpleParabolicEllipticSolver` essentially inherits from *both* `AbstractDynamicPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

If you have linear, coupled (see later) set of PDEs and can write the discretisation in this form, it is very easy to write a solver using this design—see above classes and other examples in the code.

Solver-assembler objects

The original Chaste design just considered such problems, and for these problems solvers don't need to own assemblers—**solvers are assemblers**. The concrete 'assembler-solver' class for a particular problem needs to implement `ComputeMatrixTerm()`, `ComputeVectorTerm()` etc. This design pattern is still used:

`SimpleLinearEllipticSolver` essentially inherits from *both* `AbstractStaticPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

`SimpleParabolicEllipticSolver` essentially inherits from *both* `AbstractDynamicPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

If you have linear, coupled (see later) set of PDEs and can write the discretisation in this form, it is very easy to write a solver using this design—see above classes and other examples in the code.

Solver-assembler objects

The original Chaste design just considered such problems, and for these problems solvers don't need to own assemblers—**solvers are assemblers**. The concrete 'assembler-solver' class for a particular problem needs to implement `ComputeMatrixTerm()`, `ComputeVectorTerm()` etc. This design pattern is still used:

`SimpleLinearEllipticSolver` essentially inherits from *both* `AbstractStaticPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

`SimpleParabolicEllipticSolver` essentially inherits from *both* `AbstractDynamicPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

If you have linear, coupled (see later) set of PDEs and can write the discretisation in this form, it is very easy to write a solver using this design—see above classes and other examples in the code.

Solver-assembler objects

The original Chaste design just considered such problems, and for these problems solvers don't need to own assemblers—**solvers are assemblers**. The concrete 'assembler-solver' class for a particular problem needs to implement `ComputeMatrixTerm()`, `ComputeVectorTerm()` etc. This design pattern is still used:

`SimpleLinearEllipticSolver` essentially inherits from *both* `AbstractStaticPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

`SimpleParabolicEllipticSolver` essentially inherits from *both* `AbstractDynamicPdeSolver` and `AbstractFeObjectAssembler<true,true>` and implements `ComputeMatrixTerm(..)` and `ComputeVectorTerm(..)`

If you have linear, coupled (see later) set of PDEs and can write the discretisation in this form, it is very easy to write a solver using this design—see above classes and other examples in the code.