Numerical methods and object-oriented design

Pras Pathmanathan

Summer 2011

Introduction

1. Write down equations to be solved
2. Discuss numerical schemes that can be used and summarise some of the important features
3. Discuss a sensible object-oriented design to implement the scheme
4. Describe the (current) Chaste implementation for reference

Introduction

1. Write down equations to be solved
2. Discuss numerical schemes that can be used and summarise some of the important features
3. Discuss a sensible object-oriented design to implement the scheme
4. Describe the (current) Chaste implementation for reference

Introduction

1. Write down equations to be solved
2. Discuss numerical schemes that can be used and summarise some of the important features
3. Discuss a sensible object-oriented design to implement the scheme
4. Describe the (current) Chaste implementation for reference

Introduction

1. Write down equations to be solved
2. Discuss numerical schemes that can be used and summarise some of the important features
3. Discuss a sensible object-oriented design to implement the scheme
4. Describe the (current) Chaste implementation for reference

Introduction

1. Intro to object-oriented programming — *lecture 1*
2. ODEs — *lecture 1*
3. Simple (linear, single-variable) PDEs and the finite element method (FEM) — *lectures 2 and 3*
4. Coupled and nonlinear PDEs — *lecture 4*
5. Cardiac electro-physiological PDEs — *lecture 4*
6. Other methods for solving PDEs — *lecture 5*
7. Continuum mechanics — *lectures 5 and 6*

Introduction

1. Intro to object-oriented programming — *lecture 1*
2. ODEs — *lecture 1*
3. Simple (linear, single-variable) PDEs and the finite element method (FEM) — *lectures 2 and 3*
4. Coupled and nonlinear PDEs — *lecture 4*
5. Cardiac electro-physiological PDEs — *lecture 4*
6. Other methods for solving PDEs — *lecture 5*
7. Continuum mechanics — *lectures 5 and 6*

Introduction

1. Intro to object-oriented programming — *lecture 1*
2. ODEs — *lecture 1*
3. Simple (linear, single-variable) PDEs and the finite element method (FEM) — *lectures 2 and 3*
4. Coupled and nonlinear PDEs — *lecture 4*
5. Cardiac electro-physiological PDEs — *lecture 4*
6. Other methods for solving PDEs — *lecture 5*
7. Continuum mechanics — *lectures 5 and 6*

Introduction

1. Intro to object-oriented programming — *lecture 1*
2. ODEs — *lecture 1*
3. Simple (linear, single-variable) PDEs and the finite element method (FEM) — *lectures 2 and 3*
4. Coupled and nonlinear PDEs — *lecture 4*
5. Cardiac electro-physiological PDEs — *lecture 4*
6. Other methods for solving PDEs — *lecture 5*
7. Continuum mechanics — *lectures 5 and 6*

Introduction

1. Intro to object-oriented programming — *lecture 1*
2. ODEs — *lecture 1*
3. Simple (linear, single-variable) PDEs and the finite element method (FEM) — *lectures 2 and 3*
4. Coupled and nonlinear PDEs — *lecture 4*
5. Cardiac electro-physiological PDEs — *lecture 4*
6. Other methods for solving PDEs — *lecture 5*
7. Continuum mechanics — *lectures 5 and 6*

Introduction

1. Intro to object-oriented programming — *lecture 1*
2. ODEs — *lecture 1*
3. Simple (linear, single-variable) PDEs and the finite element method (FEM) — *lectures 2 and 3*
4. Coupled and nonlinear PDEs — *lecture 4*
5. Cardiac electro-physiological PDEs — *lecture 4*
6. Other methods for solving PDEs — *lecture 5*
7. Continuum mechanics — *lectures 5 and 6*

Introduction

Things that are not part of the course

- C++
- Specific design decisions in Chaste
- Solving linear systems

Object Oriented Programming

Classes

The basic data-types in standard programming are integers, floating point real numbers, boolean flags, etc.

Object-oriented programming is based on user-defined complex data-types, known as **classes**, representing, for example: Mesh, Cat, Measurement, PdeSolver, ..

Classes are composed of **data** (**member variables**) and **methods** (functions)

Classes can be considered to be a collection of related data, with functions for using the data appropriately.

Classes - example

For example, consider the following simple class for representing a 'human'

```
class Human:
    Data:
        mAge (an integer)
    Methods:
        SetAge(age)
        GetAge()
```

The usage could be something like

```
Human ozzy;
ozzy.SetAge(age);

Human miguel;
..
if(ozzy.GetAge() < miguel.GetAge())
  ..
```

Objects are **instantiations** of classes - in the above example 'Human' is a class, 'ozzy' and 'miguel' are objects.

Classes - example

For example, consider the following simple class for representing a 'human'

```
class Human:
    Data:
        mAge (an integer)
    Methods:
        SetAge(age)
        GetAge()
```

The usage could be something like

```
Human ozzy;
ozzy.SetAge(age);

Human miguel;
..
if(ozzy.GetAge() < miguel.GetAge())
 ..
```

Objects are **instantiations** of classes - in the above example 'Human' is a class, 'ozzy' and 'miguel' are objects.

Classes - inheritance

Suppose we want to write a class for an ‘academic’. We don’t want to have to copy all the code relating to the fact that academics are (usually) humans. **Inheritance** gets around this

```
class Academic inherits from Human:
    Data:
        mNumPapers
    Methods:
        PublishPaper()                 (increments mNumPapers by one)
        GetNumPapers()
```

Classes - inheritance

Example usage:

```
Academic hawking;
..
if(hawking.GetAge()<30 && hawking.GetNumPapers()>30)
  ..
```

- The original class (`Human`) is referred to as the parent class / superclass / base class
- The inheriting class (`Academic`) is referred to as the child class / subclass / derived class.

Abstract classes

Abstract classes are classes that contain an abstract (or ‘pure virtual’) method. These are methods which are declared but not implemented.

```
class AbstractAnimal:
    Data:
        mIsHungry
    Methods:
        Eat()                                  (set mIsHungry to false)
        MakeNoise()            (Abstract method, implementation not given)
```

Abstract classes **cannot be instantiated**, i.e. the following is not allowed

```
AbstractAnimal rover
```

Instead, a subclass must be written which implements the abstract method..

Abstract classes

Example ‘concrete classes’, inheriting from `AbstractAnimal`:

```
class Dog inherits from AbstractAnimal:
    Methods:
        MakeNoise()                         (print out 'woof')
```

```
class Cat inherits from AbstractAnimal:
    Methods:
        MakeNoise()                         (print out 'meow')
```

As these have implemented the abstract methods, they can be instantiated:

```
Cat scratchy;
Dog brian;
scratchy.MakeNoise();
brian.MakeNoise();
```

Abstract classes

Example 'concrete classes', inheriting from `AbstractAnimal`:

```
class Dog inherits from AbstractAnimal:
    Methods:
        MakeNoise()                          (print out 'woof')
```

```
class Cat inherits from AbstractAnimal:
    Methods:
        MakeNoise()                          (print out 'meow')
```

As these have implemented the abstract methods, they can be instantiated:

```
Cat scratchy;
Dog brian;
scratchy.MakeNoise();
brian.MakeNoise();
```

Abstract classes

The following isn't be very neat

```
class Human:
    Methods:
        ..
        SetPetDog(dog)
        SetPetCat(cat)
```

Instead, we can do

```
class Human:
    Methods:
        ..
        SetPet(abstractAnimal)
```

Inside `SetPet()` we could call `MakeNoise()` on the abstract animal and the program would decide *at runtime* which is the appropriate function to run.

Colour scheme

`AbstractAnimal`:

Member var: `mIsHungry`
Method: `Eat()`
Abs. method: `MakeNoise()`

`Dog`: *inherits from* `AbstractAnimal`
Implemented method: `MakeNoise()`

`Cat`: *inherits from* `AbstractAnimal`
Implemented method: `MakeNoise()`

Solving ODEs

The forward and backward Euler methods

Consider the system of ODEs:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = f(t, \mathbf{y})$$

with initial condition $\mathbf{y}(0) = \mathbf{y}_0$. Given a timestep Δt, we require a numerical approximation $\mathbf{y}^0 (= \mathbf{y}_0), \mathbf{y}^1, \mathbf{y}^2, \dots$ Here $\mathbf{y}^n$ represents the numerical solution at time $t^n = n\Delta t$.

The **forward Euler discretisation** is

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = f(t^n, \mathbf{y}^n) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\, f(t^n, \mathbf{y}^n)$$

which explicitly gives each $\mathbf{y}^{n+1}$ in terms of $\mathbf{y}^n$, i.e. this is an **explicit** scheme.

The **backward Euler discretisation** is

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = f(t^{n+1}, \mathbf{y}^{n+1}) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} - \Delta t\, f(t^{n+1}, \mathbf{y}^{n+1}) = \mathbf{y}^n$$

which in general is a nonlinear system of equations for $\mathbf{y}^{n+1}$, i.e. an **implicit** scheme.

The forward and backward Euler methods

Consider the system of ODEs:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = f(t, \mathbf{y})$$

with initial condition $\mathbf{y}(0) = \mathbf{y}_0$. Given a timestep Δt, we require a numerical approximation $\mathbf{y}^0 (= \mathbf{y}_0), \mathbf{y}^1, \mathbf{y}^2, \ldots$ Here $\mathbf{y}^n$ represents the numerical solution at time $t^n = n\Delta t$.

The **forward Euler discretisation** is

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = f(t^n, \mathbf{y}^{n}) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\, f(t^n, \mathbf{y}^{n})$$

which explicitly gives each $\mathbf{y}^{n+1}$ in terms of $\mathbf{y}^n$, i.e. this is an **explicit** scheme.

The **backward Euler discretisation** is

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = f(t^{n+1}, \mathbf{y}^{n+1}) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} - \Delta t\, f(t^{n+1}, \mathbf{y}^{n+1}) = \mathbf{y}^n$$

which in general is a nonlinear system of equations for $\mathbf{y}^{n+1}$, i.e. an **implicit** scheme.

The forward and backward Euler methods

Consider the system of ODEs:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = f(t, \mathbf{y})$$

with initial condition $\mathbf{y}(0) = \mathbf{y}_0$. Given a timestep Δt, we require a numerical approximation $\mathbf{y}^0 (= \mathbf{y}_0), \mathbf{y}^1, \mathbf{y}^2, \ldots$ Here $\mathbf{y}^n$ represents the numerical solution at time $t^n = n\Delta t$.

The **forward Euler discretisation** is

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = f(t^n, \mathbf{y}^{n}) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\, f(t^n, \mathbf{y}^{n})$$

which explicitly gives each $\mathbf{y}^{n+1}$ in terms of $\mathbf{y}^n$, i.e. this is an **explicit** scheme.

The **backward Euler discretisation** is

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^n}{\Delta t} = f(t^{n+1}, \mathbf{y}^{n+1}) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} - \Delta t\, f(t^{n+1}, \mathbf{y}^{n+1}) = \mathbf{y}^n$$

which in general is a nonlinear system of equations for $\mathbf{y}^{n+1}$, i.e. an **implicit** scheme.

Backward Euler

Backward Euler:

$$\frac{\mathbf{y}^{n+1} - \mathbf{y}^{n}}{\Delta t} = f(t^{n+1}, \mathbf{y}^{n+1}) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} - \Delta t\, f(t^{n+1}, \mathbf{y}^{n+1}) = \mathbf{y}^{n}$$

This is a nonlinear equation if f is nonlinear, and a linear system if f is linear and multi-dimensional (better than nonlinear, worse than explicit).

For example:

- 1 unknown, satisfying equation $\frac{\mathrm{d}y}{\mathrm{d}t} = e^{-y}$: the discretisation is

$$y^{n+1} - \Delta t\, e^{-y^{n+1}} = y^{n}$$

- linear set of M ODEs $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = A\mathbf{y}$: the discretisation is

$$(I - \Delta t\, A)\mathbf{y}^{n+1} = \mathbf{y}^{n}$$

Backward Euler

Backward Euler:

$$\frac{\mathbf{y}^{n+1}-\mathbf{y}^{n}}{\Delta t} = f(t^{n+1},\mathbf{y}^{n+1}) \qquad \Rightarrow \qquad \mathbf{y}^{n+1} - \Delta t\, f(t^{n+1},\mathbf{y}^{n+1}) = \mathbf{y}^{n}$$

This is a nonlinear equation if f is nonlinear, and a linear system if f is linear and multi-dimensional (better than nonlinear, worse than explicit).

For example:

- 1 unknown, satisfying equation $\frac{\mathrm{d}y}{\mathrm{d}t} = e^{-y}$: the discretisation is

$$y^{n+1} - \Delta t\, e^{-y^{n+1}} = y^{n}$$

- linear set of M ODEs $\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = A\mathbf{y}$: the discretisation is

$$(I - \Delta t\, A)\mathbf{y}^{n+1} = \mathbf{y}^{n}$$

Accuracy

Write an explicit one-step method as: $y^{n+1} = y^n + \Delta t\, \phi(t^n, y^n; \Delta t)$

Truncation error

The truncation error is defined as

$$T^n = y(t^{n+1}) - y(t^n) - \Delta t\, \phi(t^n, y(t_n); \Delta t);$$

or, equivalently: **if** $y(t^n) = y^n$, then

$$T^n = y(t^{n+1}) - y^{n+1}$$

i.e. it is the local error induced in a single timestep. Note:

- for implicit/multi-step methods there is an analogous definition.
- some definitions divide through by Δt

It is easy to show using a Taylor expansion that

$$T^n = \mathcal{O}(\Delta t^2)$$

for both forward and backward Euler.

Accuracy

Write an explicit one-step method as: $y^{n+1} = y^n + \Delta t\, \phi(t^n, y^n; \Delta t)$

Truncation error

The truncation error is defined as

$$T^n = y(t^{n+1}) - y(t^n) - \Delta t\, \phi(t^n, y(t_n); \Delta t);$$

or, equivalently: **if** $y(t^n) = y^n$, then

$$T^n = y(t^{n+1}) - y^{n+1}$$

i.e. it is the local error induced in a single timestep. Note:

- for implicit/multi-step methods there is an analogous definition.
- some definitions divide through by Δt

It is easy to show using a Taylor expansion that

$$T^n = \mathcal{O}(\Delta t^2)$$

for both forward and backward Euler.

Accuracy

Write an explicit one-step method as: $y^{n+1} = y^n + \Delta t\, \phi(t^n, y^n; \Delta t)$

Truncation error

The truncation error is defined as

$$T^n = y(t^{n+1}) - y(t^n) - \Delta t\, \phi(t^n, y(t_n); \Delta t);$$

or, equivalently: **if** $y(t^n) = y^n$, then

$$T^n = y(t^{n+1}) - y^{n+1}$$

i.e. it is the local error induced in a single timestep. Note:

- for implicit/multi-step methods there is an analogous definition.
- some definitions divide through by Δt

It is easy to show using a Taylor expansion that

$$T^n = \mathcal{O}(\Delta t^2)$$

for both forward and backward Euler.

Accuracy

Global error

The global error is simply defined as

$$e_n = y(t^n) - y^n$$

We only consider global error on some fixed time interval $[0, T_{\mathrm{end}}]$. Let $T_{\mathrm{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\mathrm{end}}\Delta t) = \mathcal{O}(\Delta t)$, and therefore that $e_n = \mathcal{O}(\Delta t)$.

This can be shown to be the case under assuming mild conditions[1] on f:

$$e_n = \mathcal{O}(\Delta t)$$

We say the forward and backward Euler methods are **first-order**

[1] Basically, $f(t, y)$ is Lipschitz continuous in y—the same conditions which are required for the existence of a unique solution of the ODE $\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y)$

Accuracy

Global error

The global error is simply defined as

$$e_n = y(t^n) - y^n$$

We only consider global error on some fixed time interval $[0, T_{\mathrm{end}}]$. Let $T_{\mathrm{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\mathrm{end}}\Delta t) = \mathcal{O}(\Delta t)$, and therefore that $e_n = \mathcal{O}(\Delta t)$.

This can be shown to be the case under assuming mild conditions[1] on f:

$$e_n = \mathcal{O}(\Delta t)$$

We say the forward and backward Euler methods are **first-order**

[1] Basically, $f(t, y)$ is Lipschitz continuous in y—the same conditions which are required for the existence of a unique solution of the ODE $\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y)$

Accuracy

Global error

The global error is simply defined as

$$e_n = y(t^n) - y^n$$

We only consider global error on some fixed time interval $[0, T_{\mathrm{end}}]$. Let $T_{\mathrm{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\mathrm{end}}\Delta t) = \mathcal{O}(\Delta t)$, and therefore that $e_n = \mathcal{O}(\Delta t)$.

This can be shown to be the case under assuming mild conditions[1] on f:

$$e_n = \mathcal{O}(\Delta t)$$

We say the forward and backward Euler methods are **first-order**

[1] Basically, $f(t, y)$ is Lipschitz continuous in y—the same conditions which are required for the existence of a unique solution of the ODE $\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y)$

Accuracy

Global error

The global error is simply defined as

$$e_n = y(t^n) - y^n$$

We only consider global error on some fixed time interval $[0, T_{\mathrm{end}}]$. Let $T_{\mathrm{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\mathrm{end}}\Delta t) = \mathcal{O}(\Delta t)$, and therefore that $e_n = \mathcal{O}(\Delta t)$.

This can be shown to be the case under assuming mild conditions[1] on f:

$$e_n = \mathcal{O}(\Delta t)$$

We say the forward and backward Euler methods are **first-order**

[1] Basically, $f(t, y)$ is Lipschitz continuous in y—the same conditions which are required for the existence of a unique solution of the ODE $\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y)$

Stability

There are various notions of stability

Zero-stability

- Zero stability is the stability of the numerical solution to changes in initial condition y^0
- This is the essentially that small errors (at any time) do not grow unbounded
- A non-zero-stable method would be useless computationally
- **Dahlquist equivalence theorem**: for a 'consistent' multistep method with 'consistent' initial values: zero-stability $\Leftrightarrow$ convergence

A-stability

Consider the ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y, \text{ with } y(0) = 1 \qquad \Rightarrow \qquad y = e^{\lambda t}$$

If $\lambda < 0$, then $y \to 0$ as $t \to \infty$.

Does the numerical solution y^n satisfy $y^n \to 0$ as $n \to \infty$, with fixed Δt?

Stability

There are various notions of stability

Zero-stability

- Zero stability is the stability of the numerical solution to changes in initial condition y^0
- This is the essentially that small errors (at any time) do not grow unbounded
- A non-zero-stable method would be useless computationally
- **Dahlquist equivalence theorem**: for a 'consistent' multistep method with 'consistent' initial values: zero-stability $\Leftrightarrow$ convergence

A-stability

Consider the ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y, \text{ with } y(0) = 1 \qquad \Rightarrow \qquad y = e^{\lambda t}$$

If $\lambda < 0$, then $y \to 0$ as $t \to \infty$.

Does the numerical solution y^n satisfy $y^n \to 0$ as $n \to \infty$, with fixed Δt?

Stability

There are various notions of stability

Zero-stability

- Zero stability is the stability of the numerical solution to changes in initial condition y^0
- This is the essentially that small errors (at any time) do not grow unbounded
- A non-zero-stable method would be useless computationally
- **Dahlquist equivalence theorem**: for a 'consistent' multistep method with 'consistent' initial values: zero-stability $\Leftrightarrow$ convergence

A-stability

Consider the ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y, \text{ with } y(0) = 1 \qquad \Rightarrow \qquad y = e^{\lambda t}$$

If $\lambda < 0$, then $y \to 0$ as $t \to \infty$.

Does the numerical solution y^n satisfy $y^n \to 0$ as $n \to \infty$, with fixed Δt?

Stability

A-stability

Consider the ODE

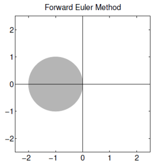
$$\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y, \text{ with } y(0) = 1 \qquad \Rightarrow \qquad y = e^{\lambda t}$$

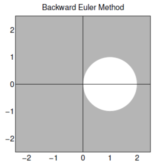
If $\lambda < 0$, then $y \to 0$ as $t \to \infty$.

Does the numerical solution y^n satisfy $y^n \to 0$ as $n \to \infty$, with fixed Δt?

If $\lambda < 0$

- Forward Euler: $y^n \to 0$ only if $\Delta t < -\frac{2}{\lambda}$, i.e. conditional stability
- Backward Euler: $y^n \to 0$ for all Δt, i.e. unconditional stability





Other discretisations

One-step:

- **Forward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\mathcal{O}(\Delta t)$
- **Backward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^{n+1}, y^{n+1})$ $\mathcal{O}(\Delta t)$
- **Trapezoidal rule**: $y^{n+1} = y^n + \frac{1}{2}\Delta t \left(f(t^n, y^n) + f(t^{n+1}, y^{n+1}) \right)$ $\mathcal{O}(\Delta t^2)$
- **Heun's method**:
  $y^{n+1} = y^n + \frac{1}{2}\Delta t \left(f(t^n, y^n) + f(t^{n+1}, y^n + \Delta t\, f(t^n, y^n)) \right)$ $\mathcal{O}(\Delta t^2)$
- **Four-stage Runge-Kutta**:
  $y^{n+1} = y^n + \frac{1}{6}\Delta t (k_1 + 2k_2 + 2k_3 + k_4)$ $\mathcal{O}(\Delta t^4)$
  where $k_1 = f(t^n, y^n)$, $k_2 = f(t^n + \frac{1}{2}\Delta t, y^n + \frac{1}{2}\Delta t\, k_1)$,...

Multi-step:

- **Simpson's Rule**: $y^{n+2} = y^{n+1} + \frac{1}{3}\Delta t \left(f^{n+2} + 4f^{n+1} + f^n \right)$ $\mathcal{O}(\Delta t^4)$
- **Adams-Bashforth**: $y^{n+2} = y^{n+1} + \frac{1}{2}\Delta t \left(3f^{n+1} - f^n \right)$ $\mathcal{O}(\Delta t^2)$

Other discretisations

One-step:

- **Forward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\quad\mathcal{O}(\Delta t)$
- **Backward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^{n+1}, y^{n+1})$ $\quad\mathcal{O}(\Delta t)$
- **Trapezoidal rule**: $y^{n+1} = y^n + \frac{1}{2}\Delta t\, \left(f(t^n, y^n) + f(t^{n+1}, y^{n+1})\right)$ $\quad\mathcal{O}(\Delta t^2)$
- **Heun's method**:
  $y^{n+1} = y^n + \frac{1}{2}\Delta t\, \left(f(t^n, y^n) + f(t^{n+1}, y^n + \Delta t\, f(t^n, y^n))\right)$ $\quad\mathcal{O}(\Delta t^2)$
- **Four-stage Runge-Kutta**:
  $y^{n+1} = y^n + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4)$ $\quad\mathcal{O}(\Delta t^4)$
  where $k_1 = f(t^n, y^n)$, $k_2 = f(t^n + \frac{1}{2}\Delta t, y^n + \frac{1}{2}\Delta t\, k_1)$,...

Multi-step:

- **Simpson's Rule**: $y^{n+2} = y^{n+1} + \frac{1}{3}\Delta t\, \left(f^{n+2} + 4f^{n+1} + f^n\right)$ $\quad\mathcal{O}(\Delta t^4)$
- **Adams-Bashforth**: $y^{n+2} = y^{n+1} + \frac{1}{2}\Delta t\, \left(3f^{n+1} - f^n\right)$ $\quad\mathcal{O}(\Delta t^2)$

Other discretisations

One-step:

- **Forward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\qquad \mathcal{O}(\Delta t)$
- **Backward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^{n+1}, y^{n+1})$ $\qquad \mathcal{O}(\Delta t)$
- **Trapezoidal rule**: $y^{n+1} = y^n + \frac{1}{2}\Delta t \left(f(t^n, y^n) + f(t^{n+1}, y^{n+1}) \right)$ $\qquad \mathcal{O}(\Delta t^2)$
- **Heun's method**:
  $y^{n+1} = y^n + \frac{1}{2}\Delta t \left(f(t^n, y^n) + f(t^{n+1}, y^n + \Delta t\, f(t^n, y^n)) \right)$ $\qquad \mathcal{O}(\Delta t^2)$
- **Four-stage Runge-Kutta**:
  $y^{n+1} = y^n + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4)$ $\qquad \mathcal{O}(\Delta t^4)$
  where $k_1 = f(t^n, y^n)$, $k_2 = f(t^n + \frac{1}{2}\Delta t, y^n + \frac{1}{2}\Delta t\, k_1), \ldots$

Multi-step:

- **Simpson's Rule**: $y^{n+2} = y^{n+1} + \frac{1}{3}\Delta t \left(f^{n+2} + 4f^{n+1} + f^n \right)$ $\qquad \mathcal{O}(\Delta t^4)$
- **Adams-Bashforth**: $y^{n+2} = y^{n+1} + \frac{1}{2}\Delta t \left(3f^{n+1} - f^n \right)$ $\qquad \mathcal{O}(\Delta t^2)$

Other discretisations

One-step:

- **Forward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\quad\mathcal{O}(\Delta t)$
- **Backward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^{n+1}, y^{n+1})$ $\quad\mathcal{O}(\Delta t)$
- **Trapezoidal rule**: $y^{n+1} = y^n + \frac{1}{2}\Delta t\left(f(t^n, y^n) + f(t^{n+1}, y^{n+1})\right)$ $\quad\mathcal{O}(\Delta t^2)$
- **Heun's method**:
  $y^{n+1} = y^n + \frac{1}{2}\Delta t\left(f(t^n, y^n) + f(t^{n+1}, y^n + \Delta t\, f(t^n, y^n))\right)$ $\quad\mathcal{O}(\Delta t^2)$
- **Four-stage Runge-Kutta**:
  $y^{n+1} = y^n + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4)$ $\quad\mathcal{O}(\Delta t^4)$
  where $k_1 = f(t^n, y^n),\ k_2 = f(t^n + \frac{1}{2}\Delta t, y^n + \frac{1}{2}\Delta t\, k_1), \dots$

Multi-step:

- **Simpson's Rule**: $y^{n+2} = y^{n+1} + \frac{1}{3}\Delta t\left(f^{n+2} + 4f^{n+1} + f^n\right)$ $\quad\mathcal{O}(\Delta t^4)$
- **Adams-Bashforth**: $y^{n+2} = y^{n+1} + \frac{1}{2}\Delta t\left(3f^{n+1} - f^n\right)$ $\quad\mathcal{O}(\Delta t^2)$

Other discretisations

One-step:

- **Forward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\qquad \mathcal{O}(\Delta t)$
- **Backward Euler**: $y^{n+1} = y^n + \Delta t\, f(t^{n+1}, y^{n+1})$ $\qquad \mathcal{O}(\Delta t)$
- **Trapezoidal rule**: $y^{n+1} = y^n + \frac{1}{2}\Delta t \left(f(t^n, y^n) + f(t^{n+1}, y^{n+1})\right)$ $\qquad \mathcal{O}(\Delta t^2)$
- **Heun's method**:
  $y^{n+1} = y^n + \frac{1}{2}\Delta t \left(f(t^n, y^n) + f(t^{n+1}, y^n + \Delta t\, f(t^n, y^n))\right)$ $\qquad \mathcal{O}(\Delta t^2)$
- **Four-stage Runge-Kutta**:
  $y^{n+1} = y^n + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4)$ $\qquad \mathcal{O}(\Delta t^4)$
  where $k_1 = f(t^n, y^n)$, $k_2 = f(t^n + \frac{1}{2}\Delta t, y^n + \frac{1}{2}\Delta t\, k_1)$,...

Multi-step:

- **Simpson's Rule**: $y^{n+2} = y^{n+1} + \frac{1}{3}\Delta t \left(f^{n+2} + 4f^{n+1} + f^n\right)$ $\qquad \mathcal{O}(\Delta t^4)$
- **Adams-Bashforth**: $y^{n+2} = y^{n+1} + \frac{1}{2}\Delta t \left(3f^{n+1} - f^n\right)$ $\qquad \mathcal{O}(\Delta t^2)$

Object-oriented implementation

A (standard) Matlab approach uses function pointers:

```
[T,Y] = ode45(@my_func,[0 1],[1 2 3]);

function dydt = my_func(t,y)
dydt = y.^2;
```

Object-oriented approach:

`AbstractOdeSystem`:

Member var: `mSize`

▷ *i.e. the dimension of the vector* **y**

Abs. method: `EvaluateYDerivatives(t, y)`

▷ *Declares the function representing* $f(t, \mathbf{y})$

`MyOdeSystem`: *inherits from* `AbstractOdeSystem`

Implemented method: `EvaluateYDerivatives(t, y)`

▷ *One particular choice of* $f(t, \mathbf{y})$

Object-oriented implementation: solvers

`AbstractOneStepOdeSolver`:
  Abs. method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

`ForwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`
  Implemented method: `Solve(..)`
  - ▷ *Implements a forward Euler solve*

`BackwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`
  Implemented method: `Solve(..)`
  - ▷ *Implements a backward Euler solve*

This isn't optimal, because the loop over time is implemented in both the solvers, but isn't specific to either

Object-oriented implementation: solvers

`AbstractOneStepOdeSolver`:
Abs. method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

`ForwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`
Implemented method: `Solve(..)`
▷ *Implements a forward Euler solve*

`BackwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`
Implemented method: `Solve(..)`
▷ *Implements a backward Euler solve*

This isn't optimal, because the loop over time is implemented in both the solvers, but isn't specific to either

Object-oriented implementation: solvers

`AbstractOneStepOdeSolver`:

Abs. method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

`ForwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `Solve(..)`

- ▷ *Implements a forward Euler solve*

`BackwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `Solve(..)`

- ▷ *Implements a backward Euler solve*

This isn't optimal, because the loop over time is implemented in both the solvers, but isn't specific to either

Object-oriented implementation: solvers

`AbstractOneStepOdeSolver`:
    Abs. method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

`ForwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`
    Implemented method: `Solve(..)`
        ▷ *Implements a forward Euler solve*

`BackwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`
    Implemented method: `Solve(..)`
        ▷ *Implements a backward Euler solve*

This isn't optimal, because the loop over time is implemented in both the solvers, but isn't specific to either

Object-oriented implementation: solvers

`AbstractOneStepOdeSolver`:

Method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

▷ *Implements a loop over time, and each timestep calls the following:*

Abs. method: `CalculateNextYValue(currentYValue)`

`ForwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

▷ *Takes in* $\mathbf{y}^n$, *returns* $\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\, f(t^n, \mathbf{y}^n)$

`BackwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

▷ *Takes in* $\mathbf{y}^n$, *solves* $\mathbf{y}^{n+1} - \Delta t\, f(t^{n+1}, \mathbf{y}^{n+1}) = \mathbf{y}^n$, *returns* $\mathbf{y}^{n+1}$

Object-oriented implementation: solvers

`AbstractOneStepOdeSolver`:

Method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

▷ *Implements a loop over time, and each timestep calls the following:*

Abs. method: `CalculateNextYValue(currentYValue)`

`ForwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

▷ *Takes in* $\mathbf{y}^n$, *returns* $\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\, f(t^n, \mathbf{y}^n)$

`BackwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

▷ *Takes in* $\mathbf{y}^n$, *solves* $\mathbf{y}^{n+1} - \Delta t\, f(t^{n+1}, \mathbf{y}^{n+1}) = \mathbf{y}^n$, *returns* $\mathbf{y}^{n+1}$

Object-oriented implementation: solvers

`AbstractOneStepOdeSolver`:

Method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

▷ *Implements a loop over time, and each timestep calls the following:*

Abs. method: `CalculateNextYValue(currentYValue)`

`ForwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

▷ *Takes in* $\mathbf{y}^n$, *returns* $\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\, f(t^n, \mathbf{y}^n)$

`BackwardEulerSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

▷ *Takes in* $\mathbf{y}^n$, *solves* $\mathbf{y}^{n+1} - \Delta t\, f(t^{n+1}, \mathbf{y}^{n+1}) = \mathbf{y}^n$, *returns* $\mathbf{y}^{n+1}$

Multiple ODE systems

A complication: for large-scale cardiac problems (and other applications), have a system of ODEs for each point in space—which leads to (for example), a system of ODEs for each node in the computational mesh.

For large simulations this is potentially millions of systems of ODEs. The current solution at each node needs to be stored.

One approach would be to store an N by M matrix, N the number of nodes, M the size of each system.

$$\begin{pmatrix} \mathbf{y}^{(\text{node } 1)} \\ \vdots \\ \mathbf{y}^{(\text{node } N)} \end{pmatrix}$$

Design for multiple ODE systems

The object-oriented approach is to store the state variables (i.e. 'current solution') in the ODE system object.

`AbstractOdeSystem`:

Member var: `mSize`

Member var: `mStateVariables` $\longleftarrow$

Abs. method: `EvaluateYDerivatives(t, y)`

The solver now has two different types of Solve method

`AbstractOneStepOdeSolver`:

Method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

▷ *Uses the given initial condition, returns computed solution, ignores state variables inside the ODE sysem*

Method: `SolveAndUpdateStateVariable(absOdeSys,t0,t1)`

▷ *Use state variables in ODE system as initial condition, puts final solution in state variables and returns nothing*

Abs. method: `CalculateNextYValue(..)`

`ForwardEulerSolver` etc are unchanged

Design for multiple ODE systems

The object-oriented approach is to store the state variables (i.e. 'current solution') in the ODE system object.

`AbstractOdeSystem`:

Member var: `mSize`

Member var: `mStateVariables` ⟵

Abs. method: `EvaluateYDerivatives(t, y)`

The solver now has two different types of Solve method

`AbstractOneStepOdeSolver`:

Method: `Solve(abstractOdeSystem,t0,t1,initialCond)`

- ▷ *Uses the given initial condition, returns computed solution, ignores state variables inside the ODE sysem*

Method: `SolveAndUpdateStateVariable(absOdeSys,t0,t1)`

- ▷ *Use state variables in ODE system as initial condition, puts final solution in state variables and returns nothing*

Abs. method: `CalculateNextYValue(..)`

`ForwardEulerSolver` etc are unchanged

ODE classes in Chaste - ODE system

See folder `ode/src/common/`

`AbstractParameterisedSystem`:

- *Member var:* `mNumberOfStateVariables`
- *Member var:* `mStateVariables`
- *Member var:* `mpSystemInfo`
  - ▷ *Data of type* `AbstractOdeSystemInformation`

`AbstractOdeSystem`: *inherits from* `AbstractParameterisedSystem`

- *Member var:* `mDefaultInitialConditions`
- *Abs. method:* `EvaluateYDerivatives(t, y)`

`AbstractOdeSystemInformation`:

- *Member var:* `mVariableNames`
- *Member var:* `mVariableUnits`

`OdeSolution`:

- ▷ *Returned by solvers, contains times and solution values*

ODE classes in Chaste - ODE system

See folder `ode/src/common/`

`AbstractParameterisedSystem`:

Member var: `mNumberOfStateVariables`
Member var: `mStateVariables`
Member var: `mpSystemInfo`

- ▷ *Data of type* `AbstractOdeSystemInformation`

`AbstractOdeSystem`: *inherits from* `AbstractParameterisedSystem`

Member var: `mDefaultInitialConditions`
Abs. method: `EvaluateYDerivatives(t, y)`

`AbstractOdeSystemInformation`:

Member var: `mVariableNames`
Member var: `mVariableUnits`

`OdeSolution`:

- ▷ *Returned by solvers, contains times and solution values*

ODE classes in Chaste - ODE system

See folder `ode/src/common/`

`AbstractParameterisedSystem`:

Member var: `mNumberOfStateVariables`
Member var: `mStateVariables`
Member var: `mpSystemInfo`

- ▷ *Data of type* `AbstractOdeSystemInformation`

`AbstractOdeSystem`: *inherits from* `AbstractParameterisedSystem`

Member var: `mDefaultInitialConditions`
Abs. method: `EvaluateYDerivatives(t, y)`

`AbstractOdeSystemInformation`:

Member var: `mVariableNames`
Member var: `mVariableUnits`

`OdeSolution`:

- ▷ *Returned by solvers, contains times and solution values*

ODE classes in Chaste - ODE system

See folder `ode/src/common/`

`AbstractParameterisedSystem`:

Member var: `mNumberOfStateVariables`
Member var: `mStateVariables`
Member var: `mpSystemInfo`

- ▷ *Data of type* `AbstractOdeSystemInformation`

`AbstractOdeSystem`: *inherits from* `AbstractParameterisedSystem`

Member var: `mDefaultInitialConditions`
Abs. method: `EvaluateYDerivatives(t, y)`

`AbstractOdeSystemInformation`:

Member var: `mVariableNames`
Member var: `mVariableUnits`

`OdeSolution`:

- ▷ *Returned by solvers, contains times and solution values*

ODE classes in Chaste - solvers

See folder `ode/src/solver/`

`AbstractIvpOdeSolver`:
- *Abs. method:* `Solve(abstractOdeSystem,t0,t1,initialCond)`
- *Abs. method:* `SolveAndUpdateStateVariable(absOdeSys,t0,t1)`

`AbstractOneStepIvpOdeSolver`: *inherits from* `AbstractIvpOdeSolver`
- *Implemented method:* `Solve(..)`
- *Implemented method:* `SolveAndUpdateStateVariable(..)`
- *Method:* `InternalSolve(..)`
- *Abs. method:* `CalculateNextYValue(..)`

`ForwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`
- *Implemented method:* `CalculateNextYValue(..)`

`BackwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`
- *Implemented method:* `CalculateNextYValue(..)`

There are also `Heun`, `RungeKutta2` and `RungeKutta4` solvers (all one-step) and a `RungeKuttaFehlberg` (Matlab's 'ode45') (inherits from `AbstractIvpOdeSolver`).

ODE classes in Chaste - solvers

See folder `ode/src/solver/`

`AbstractIvpOdeSolver`:
Abs. method: `Solve(abstractOdeSystem,t0,t1,initialCond)`
Abs. method: `SolveAndUpdateStateVariable(absOdeSys,t0,t1)`

`AbstractOneStepIvpOdeSolver`: *inherits from* `AbstractIvpOdeSolver`
Implemented method: `Solve(..)`
Implemented method: `SolveAndUpdateStateVariable(..)`
Method: `InternalSolve(..)`
Abs. method: `CalculateNextYValue(..)`

`ForwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`
Implemented method: `CalculateNextYValue(..)`

`BackwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`
Implemented method: `CalculateNextYValue(..)`

There are also `Heun`, `RungeKutta2` and `RungeKutta4` solvers (all one-step) and a `RungeKuttaFehlberg` (Matlab's 'ode45') (inherits from `AbstractIvpOdeSolver`).

ODE classes in Chaste - solvers

See folder `ode/src/solver/`

`AbstractIvpOdeSolver`:

Abs. method: `Solve(abstractOdeSystem,t0,t1,initialCond)`
Abs. method: `SolveAndUpdateStateVariable(absOdeSys,t0,t1)`

`AbstractOneStepIvpOdeSolver`: *inherits from* `AbstractIvpOdeSolver`

Implemented method: `Solve(..)`
Implemented method: `SolveAndUpdateStateVariable(..)`
Method: `InternalSolve(..)`
Abs. method: `CalculateNextYValue(..)`

`ForwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

`BackwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

There are also `Heun`, `RungeKutta2` and `RungeKutta4` solvers (all one-step), and a `RungeKuttaFehlberg` (Matlab's 'ode45') (inherits from `AbstractIvpOdeSolver`).

ODE classes in Chaste - solvers

See folder `ode/src/solver/`

`AbstractIvpOdeSolver`:
- *Abs. method:* `Solve(abstractOdeSystem,t0,t1,initialCond)`
- *Abs. method:* `SolveAndUpdateStateVariable(absOdeSys,t0,t1)`

`AbstractOneStepIvpOdeSolver`: *inherits from* `AbstractIvpOdeSolver`
- *Implemented method:* `Solve(..)`
- *Implemented method:* `SolveAndUpdateStateVariable(..)`
- *Method:* `InternalSolve(..)`
- *Abs. method:* `CalculateNextYValue(..)`

`ForwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`
- *Implemented method:* `CalculateNextYValue(..)`

`BackwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`
- *Implemented method:* `CalculateNextYValue(..)`

There are also `Heun`, `RungeKutta2` and `RungeKutta4` solvers (all one-step), and a `RungeKuttaFehlberg` (Matlab's 'ode45') (inherits from `AbstractIvpOdeSolver`).

ODE classes in Chaste - solvers

See folder `ode/src/solver/`

`AbstractIvpOdeSolver`:

Abs. method: `Solve(abstractOdeSystem,t0,t1,initialCond)`
Abs. method: `SolveAndUpdateStateVariable(absOdeSys,t0,t1)`

`AbstractOneStepIvpOdeSolver`: *inherits from* `AbstractIvpOdeSolver`

Implemented method: `Solve(..)`
Implemented method: `SolveAndUpdateStateVariable(..)`
Method: `InternalSolve(..)`
Abs. method: `CalculateNextYValue(..)`

`ForwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

`BackwardEulerIvpSolver`: *inherits from* `AbstractOneStepOdeSolver`

Implemented method: `CalculateNextYValue(..)`

There are also `Heun`, `RungeKutta2` and `RungeKutta4` solvers (all one-step), and a `RungeKuttaFehlberg` (Matlab's 'ode45') (inherits from `AbstractIvpOdeSolver`).

Solving simple PDEs using the finite element method

FEM for simple PDEs: elliptic and parabolic linear PDEs

PDEs

Second-order PDEs commonly arise in physical models. There are three archetypal second-order PDEs

1. Elliptic PDEs, for example, Poisson's equation $\nabla^2 u + f = 0$
2. Parabolic PDEs, for example, the heat equation $u_t = \nabla^2 u + f$
3. Hyperbolic PDEs, for example, the wave equation $u_{tt} = \nabla^2 u$

PDEs

Second-order PDEs commonly arise in physical models. There are three archetypal second-order PDEs

1. **Elliptic PDEs, for example, Poisson's equation** $\nabla^2 u + f = 0$
2. **Parabolic PDEs, for example, the heat equation** $u_t = \nabla^2 u + f$
3. Hyperbolic PDEs, for example, the wave equation $u_{tt} = \nabla^2 u$

Defining PDEs in an object oriented manner

First, an abstract class defining a general linear elliptic PDE $\nabla \cdot D\boldsymbol{\nabla} u = f$, where D is a matrix-valued function of position (the diffusion tensor):

```
AbstractLinearEllipticPde:
    Abs. method: GetDiffusionTensor(x)
    Abs. method: GetForceTerm(x)
```

```
MyEllipticPde:   inherits from AbstractLinearEllipticPde
    Implemented method: GetDiffusionTensor(x)
    Implemented method: GetForceTerm(x)
```

For example $\nabla^2 u = 0$

```
LaplacesEquation:   inherits from AbstractLinearEllipticPde
    Implemented method: GetDiffusionTensor(x)
        ▷ return identity matrix
    Implemented method: GetForceTerm(x)
        ▷ return zero
```

Defining PDEs in an object oriented manner

First, an abstract class defining a general linear elliptic PDE $\nabla \cdot D\boldsymbol{\nabla} u = f$, where D is a matrix-valued function of position (the diffusion tensor):

```
AbstractLinearEllipticPde:
    Abs. method: GetDiffusionTensor(x)
    Abs. method: GetForceTerm(x)

MyEllipticPde:   inherits from AbstractLinearEllipticPde
    Implemented method: GetDiffusionTensor(x)
    Implemented method: GetForceTerm(x)
```

For example $\nabla^2 u = 0$

```
LaplacesEquation:   inherits from AbstractLinearEllipticPde
    Implemented method: GetDiffusionTensor(x)
        ▷ return identity matrix
    Implemented method: GetForceTerm(x)
        ▷ return zero
```

Defining PDEs in an object oriented manner

First, an abstract class defining a general linear elliptic PDE $\nabla \cdot D\boldsymbol{\nabla} u = f$, where D is a matrix-valued function of position (the diffusion tensor):

```
AbstractLinearEllipticPde:
    Abs. method: GetDiffusionTensor(x)
    Abs. method: GetForceTerm(x)

MyEllipticPde:   inherits from AbstractLinearEllipticPde
    Implemented method: GetDiffusionTensor(x)
    Implemented method: GetForceTerm(x)
```

For example $\nabla^2 u = 0$

```
LaplacesEquation:   inherits from AbstractLinearEllipticPde
    Implemented method: GetDiffusionTensor(x)
        ▷ return identity matrix
    Implemented method: GetForceTerm(x)
        ▷ return zero
```

Defining PDEs in an object oriented manner

Next, an abstract class defining a general linear parabolic PDE

$$\alpha u_t = \nabla \cdot D \boldsymbol{\nabla} u + f$$

where α, D and f are functions of space and time.

```
AbstractLinearParabolicPde:
    Abs. method: GetDuDtCoefficientTerm(t,x)
    Abs. method: GetDiffusionTensor(t,x)
    Abs. method: GetForceTerm(t,x)
```

For example $u_t = \nabla^2 u$

```
HeatEquation:   inherits from AbstractLinearParabolicPde
    Implemented method: GetDuDtCoefficientTerm(t,x)
        ▷ return 1
    Implemented method: GetDiffusionTensor(x)
        ▷ return identity matrix
    Implemented method: GetForceTerm(x)
        ▷ return zero
```

Defining PDEs in an object oriented manner

Next, an abstract class defining a general linear parabolic PDE

$$\alpha u_t = \nabla \cdot D \boldsymbol{\nabla} u + f$$

where α, D and f are functions of space and time.

```
AbstractLinearParabolicPde:
    Abs. method: GetDuDtCoefficientTerm(t,x)
    Abs. method: GetDiffusionTensor(t,x)
    Abs. method: GetForceTerm(t,x)
```

For example $u_t = \nabla^2 u$

```
HeatEquation:   inherits from AbstractLinearParabolicPde
    Implemented method: GetDuDtCoefficientTerm(t,x)
        ▷ return 1
    Implemented method: GetDiffusionTensor(x)
        ▷ return identity matrix
    Implemented method: GetForceTerm(x)
        ▷ return zero
```

FEM for simple PDEs: introduction to FEM

The finite element method

Stages

1. Convert equation from **strong form** to **weak form**
2. Convert infinite-dimensional problem into a finite dimensional one
3. Set up the finite element linear system to be solved

Weak form of Poisson's equation

Consider Poisson's equation:

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

subject to boundary conditions

$$\begin{aligned} u &= 0 && \text{on } \Gamma_1 \\ \boldsymbol{\nabla} u \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

Weak form

Multiply by a test function v satisfying $v = 0$ on Γ_1, and integrate:

$$\begin{aligned} v\left(\nabla^2 u\right) &= -fv \\ \int_\Omega v\left(\nabla^2 u\right)\,\mathrm{d}V &= -\int_\Omega fv\,\mathrm{d}V \\ \int_{\partial\Omega} v\,(\boldsymbol{\nabla} u \cdot \mathbf{n})\,\mathrm{d}S - \int_\Omega \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v\,\mathrm{d}V &= -\int_\Omega fv\,\mathrm{d}V \\ \int_\Omega \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v\,\mathrm{d}V &= \int_\Omega fv\,\mathrm{d}V + \int_{\Gamma_2} gv\,\mathrm{d}S \end{aligned}$$

Weak form of Poisson's equation

Consider Poisson's equation:

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

subject to boundary conditions

$$\begin{aligned} u &= 0 \qquad \text{on } \Gamma_1 \\ \boldsymbol{\nabla} u \cdot \mathbf{n} &= g \qquad \text{on } \Gamma_2 \end{aligned}$$

Weak form

Multiply by a test function v satisfying $v = 0$ on Γ_1, and integrate:

$$\begin{aligned} v\left(\nabla^2 u\right) &= -fv \\ \int_\Omega v\left(\nabla^2 u\right)\,\mathrm{d}V &= -\int_\Omega fv\,\mathrm{d}V \\ \int_{\partial\Omega} v\,(\boldsymbol{\nabla} u \cdot \mathbf{n})\,\mathrm{d}S - \int_\Omega \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v\,\mathrm{d}V &= -\int_\Omega fv\,\mathrm{d}V \\ \int_\Omega \boldsymbol{\nabla} u \cdot \boldsymbol{\nabla} v\,\mathrm{d}V &= \int_\Omega fv\,\mathrm{d}V + \int_{\Gamma_2} gv\,\mathrm{d}S \end{aligned}$$

Weak form of Poisson's equation

Let $\mathcal{V}$ be the space of all differentiable functions on Ω (more precisely, $\mathcal{V}$ is the Sobolev space $H^1(\Omega)$). Let

$$\mathcal{V}_0 = \{v \in \mathcal{V} \,:\, v = 0 \text{ on } \Gamma_1\}$$

Weak form

Find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}V = \int_\Omega fv \,\mathrm{d}V + \int_{\Gamma_2} gv \,\mathrm{d}S \qquad \forall v \in \mathcal{V}_0$$

Example

Solve $\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = 1$, $u(0) = u(1) = 0$ vs Find differentiable u satisfying $u(0) = u(1) = 0$ and: $\int_0^1 \frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x}\,\mathrm{d}x = -\int_0^1 v\,\mathrm{d}x$ for all v s.t. $v(0) = v(1) = 0$

Weak form of Poisson's equation

Let $\mathcal{V}$ be the space of all differentiable functions on Ω (more precisely, $\mathcal{V}$ is the Sobolev space $H^1(\Omega)$). Let

$$\mathcal{V}_0 = \{v \in \mathcal{V} \; : \; v = 0 \text{ on } \Gamma_1\}$$

Weak form

Find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}V = \int_\Omega f v \, \mathrm{d}V + \int_{\Gamma_2} g v \, \mathrm{d}S \qquad \forall v \in \mathcal{V}_0$$

Example

Solve $\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = 1$, $u(0) = u(1) = 0$

vs

Find differentiable u satisfying $u(0) = u(1) = 0$ and: $\int_0^1 \frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x}\,\mathrm{d}x = -\int_0^1 v \,\mathrm{d}x$ for all v s.t. $v(0) = v(1) = 0$

Weak form of Poisson's equation

Let $\mathcal{V}$ be the space of all differentiable functions on Ω (more precisely, $\mathcal{V}$ is the Sobolev space $H^1(\Omega)$). Let

$$\mathcal{V}_0 = \{v \in \mathcal{V} \,:\, v = 0 \text{ on } \Gamma_1\}$$

Weak form

Find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}V = \int_\Omega fv \,\mathrm{d}V + \int_{\Gamma_2} gv \,\mathrm{d}S \qquad \forall v \in \mathcal{V}_0$$

Example

Solve $\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} = 1$, $u(0) = u(1) = 0$

vs

Find differentiable u satisfying $u(0) = u(1) = 0$ and: $\int_0^1 \frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x}\,\mathrm{d}x = -\int_0^1 v\,\mathrm{d}x$ for all v s.t. $v(0) = v(1) = 0$

FEM discretisation

Find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}V = \int_\Omega f v \,\mathrm{d}V + \int_{\Gamma_2} g v \,\mathrm{d}S \qquad \text{for all } v \in \mathcal{V}_0$$

Take

$$\mathcal{V}_0^h = \operatorname{span}\{\phi_1, \phi_2\}$$

(where ϕ_1, ϕ_2 satisfy the Dirichlet boundary conditions), so

$$u_h = \alpha\phi_1 + \beta\phi_2$$

Linear system:

$$\begin{bmatrix} \int_\Omega \nabla\phi_1 \cdot \nabla\phi_1 \,\mathrm{d}V & \int_\Omega \nabla\phi_1 \cdot \nabla\phi_2 \,\mathrm{d}V \\ \int_\Omega \nabla\phi_2 \cdot \nabla\phi_1 \,\mathrm{d}V & \int_\Omega \nabla\phi_2 \cdot \nabla\phi_2 \,\mathrm{d}V \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \int_\Omega f\phi_1 \,\mathrm{d}V + \int_{\Gamma_2} g\phi_1 \,\mathrm{d}S \\ \int_\Omega f\phi_2 \,\mathrm{d}V + \int_{\Gamma_2} g\phi_2 \,\mathrm{d}S \end{bmatrix}$$

FEM discretisation

Find $u_h \in \mathcal{V}_0^h$ satisfying

$$\int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}V = \int_\Omega f v \,\mathrm{d}V + \int_{\Gamma_2} g v \,\mathrm{d}S \qquad \text{for all } v \in \mathcal{V}_0^h$$

Take

$$\mathcal{V}_0^h = \mathrm{span}\{\phi_1, \phi_2\}$$

(where ϕ_1, ϕ_2 satisfy the Dirichlet boundary conditions), so

$$u_h = \alpha\phi_1 + \beta\phi_2$$

Linear system:

$$\begin{bmatrix} \int_\Omega \nabla\phi_1 \cdot \nabla\phi_1 \,\mathrm{d}V & \int_\Omega \nabla\phi_1 \cdot \nabla\phi_2 \,\mathrm{d}V \\ \int_\Omega \nabla\phi_2 \cdot \nabla\phi_1 \,\mathrm{d}V & \int_\Omega \nabla\phi_2 \cdot \nabla\phi_2 \,\mathrm{d}V \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \int_\Omega f\phi_1 \,\mathrm{d}V + \int_{\Gamma_2} g\phi_1 \,\mathrm{d}S \\ \int_\Omega f\phi_2 \,\mathrm{d}V + \int_{\Gamma_2} g\phi_2 \,\mathrm{d}S \end{bmatrix}$$

FEM discretisation

Find $u_h \in \mathcal{V}_0^h$ satisfying

$$\int_\Omega \nabla u_h \cdot \nabla v \, \mathrm{d}V = \int_\Omega f v \, \mathrm{d}V + \int_{\Gamma_2} g v \, \mathrm{d}S \qquad \text{for all } v \in \mathcal{V}_0^h$$

Take

$$\mathcal{V}_0^h = \mathrm{span}\{\phi_1, \phi_2\}$$

(where ϕ_1, ϕ_2 satisfy the Dirichlet boundary conditions), so

$$u_h = \alpha\phi_1 + \beta\phi_2$$

Linear system:

$$\begin{bmatrix} \int_\Omega \nabla\phi_1 \cdot \nabla\phi_1 \, \mathrm{d}V & \int_\Omega \nabla\phi_1 \cdot \nabla\phi_2 \, \mathrm{d}V \\ \int_\Omega \nabla\phi_2 \cdot \nabla\phi_1 \, \mathrm{d}V & \int_\Omega \nabla\phi_2 \cdot \nabla\phi_2 \, \mathrm{d}V \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \int_\Omega f\phi_1 \, \mathrm{d}V + \int_{\Gamma_2} g\phi_1 \, \mathrm{d}S \\ \int_\Omega f\phi_2 \, \mathrm{d}V + \int_{\Gamma_2} g\phi_2 \, \mathrm{d}S \end{bmatrix}$$

FEM discretisation

Find $u_h \in \mathcal{V}_0^h$ satisfying

$$\int_\Omega \nabla u_h \cdot \nabla \phi_j \, \mathrm{d}V = \int_\Omega f \phi_j \, \mathrm{d}V + \int_{\Gamma_2} g \phi_j \, \mathrm{d}S \qquad \text{for } j = 1, 2$$

Take

$$\mathcal{V}_0^h = \mathrm{span}\{\phi_1, \phi_2\}$$

(where ϕ_1, ϕ_2 satisfy the Dirichlet boundary conditions), so

$$u_h = \alpha\phi_1 + \beta\phi_2$$

Linear system:

$$\begin{bmatrix} \int_\Omega \nabla\phi_1 \cdot \nabla\phi_1 \, \mathrm{d}V & \int_\Omega \nabla\phi_1 \cdot \nabla\phi_2 \, \mathrm{d}V \\ \int_\Omega \nabla\phi_2 \cdot \nabla\phi_1 \, \mathrm{d}V & \int_\Omega \nabla\phi_2 \cdot \nabla\phi_2 \, \mathrm{d}V \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \int_\Omega f\phi_1 \, \mathrm{d}V + \int_{\Gamma_2} g\phi_1 \, \mathrm{d}S \\ \int_\Omega f\phi_2 \, \mathrm{d}V + \int_{\Gamma_2} g\phi_2 \, \mathrm{d}S \end{bmatrix}$$

FEM discretisation

Find $u_h \in \mathcal{V}_0^h$ satisfying

$$\int_\Omega \nabla u_h \cdot \nabla \phi_j \, \mathrm{d}V = \int_\Omega f \phi_j \, \mathrm{d}V + \int_{\Gamma_2} g \phi_j \, \mathrm{d}S \qquad \text{for } j = 1, 2$$

Take

$$\mathcal{V}_0^h = \operatorname{span}\{\phi_1, \phi_2\}$$

(where ϕ_1, ϕ_2 satisfy the Dirichlet boundary conditions), so

$$u_h = \alpha \phi_1 + \beta \phi_2$$

Linear system:

$$\begin{bmatrix} \int_\Omega \nabla \phi_1 \cdot \nabla \phi_1 \, \mathrm{d}V & \int_\Omega \nabla \phi_1 \cdot \nabla \phi_2 \, \mathrm{d}V \\ \int_\Omega \nabla \phi_2 \cdot \nabla \phi_1 \, \mathrm{d}V & \int_\Omega \nabla \phi_2 \cdot \nabla \phi_2 \, \mathrm{d}V \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \int_\Omega f \phi_1 \, \mathrm{d}V + \int_{\Gamma_2} g \phi_1 \, \mathrm{d}S \\ \int_\Omega f \phi_2 \, \mathrm{d}V + \int_{\Gamma_2} g \phi_2 \, \mathrm{d}S \end{bmatrix}$$

FEM discretisations

Take

$$V_h = \mathrm{span}\{\phi_1, \phi_2, \ldots, \phi_N\}$$

(satisfying $\phi_j = 0$ on Γ_1) so

$$u_h = \alpha_1\phi_1 + \ldots + \alpha_N\phi_N$$

Let the stiffness matrix and RHS vector be given by

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

and solve

$$K \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix} = \mathbf{b}$$

Basis functions

FEM discretisations

Let

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

FEM discretisations

Laplace's equation: $\nabla^2 u + f = 0 \quad \rightarrow \quad K\mathbf{U} = \mathbf{b}$

Heat equation:

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \quad \rightarrow \quad M\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t} + K\mathbf{U} = \mathbf{b}$$

Time-discretised heat equation:

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla^2 u^{n+1} + f^{n+1} \quad \rightarrow \quad M\mathbf{U}^{n+1} + \Delta t\, K\mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\,\mathbf{b}^{n+1}$$

Anisotropic diffusion

Suppose we have an anisotropic diffusion tensor D (symmetric, positive definite), for example, in Poisson's equation:

$$\nabla \cdot (D\boldsymbol{\nabla} u) + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= 0 && \text{on } \Gamma_1 \\ (D\boldsymbol{\nabla} u)\cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

The weak form is: find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega (D\boldsymbol{\nabla} u)\cdot \boldsymbol{\nabla} v \,\mathrm{d}V = \int_\Omega f v \,\mathrm{d}V + \int_{\Gamma_2} g v \,\mathrm{d}S \qquad \forall v \in \mathcal{V}_0$$

and the only change in the FEM discretisation is that the stiffness matrix becomes

$$K_{jk} = \int_\Omega \boldsymbol{\nabla}\phi_j \cdot (D\boldsymbol{\nabla}\phi_k)\,\mathrm{d}V$$

Anisotropic diffusion

Suppose we have an anisotropic diffusion tensor D (symmetric, positive definite), for example, in Poisson's equation:

$$\nabla \cdot (D \boldsymbol{\nabla} u) + f = 0$$

subject to boundary conditions

$$\begin{aligned} u &= 0 \qquad \text{on } \Gamma_1 \\ (D \boldsymbol{\nabla} u) \cdot \mathbf{n} &= g \qquad \text{on } \Gamma_2 \end{aligned}$$

The weak form is: find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega (D \boldsymbol{\nabla} u) \cdot \boldsymbol{\nabla} v \, \mathrm{d}V = \int_\Omega f v \, \mathrm{d}V + \int_{\Gamma_2} g v \, \mathrm{d}S \qquad \forall v \in \mathcal{V}_0$$

and the only change in the FEM discretisation is that the stiffness matrix becomes

$$K_{jk} = \int_\Omega \boldsymbol{\nabla} \phi_j \cdot (D \boldsymbol{\nabla} \phi_k) \, \mathrm{d}V$$

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 $U_1 = c$

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

FEM stages

Solve:

$$\nabla \cdot (D \boldsymbol{\nabla} u) + f = 0$$

subject to boundary conditions

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

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

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

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

FEM for simple PDEs: FEM details

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. Also, assume we are using linear basis functions.

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. Also, assume we are using linear basis functions.

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. Also, assume we are using linear basis functions.

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. Also, assume we are using linear basis functions.

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. Also, assume we are using linear basis functions.

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 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 $(0, 0)$, $(0, 1)$, $(1, 0)$.

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

$$\begin{aligned} N_1(\xi, \eta) &= 1 - \xi - \eta \\ N_2(\xi, \eta) &= \xi \\ N_3(\xi, \eta) &= \eta \end{aligned}$$

Computing an elemental contribution

We now need to be able to compute

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

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

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

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

We now need to be able to compute

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

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

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

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

We now need to be able to compute

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

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

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

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 - the general case

Suppose we want to compute

$$\int_{\mathcal{K}} \mathcal{F}(x, y, u, \phi_1, \phi_2, \phi_3, \boldsymbol{\nabla}\phi_1, \boldsymbol{\nabla}\phi_2, \boldsymbol{\nabla}\phi_3)\,\mathrm{d}x\mathrm{d}y$$

We map to the reference element:

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

and then use **numerical quadrature**, which means f just has to be evaluated at the quadrature points.

FEM stages - full algorithm

Solve:

$$\nabla \cdot (D \boldsymbol{\nabla} u) + f = 0$$

subject to boundary conditions

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

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

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

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

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

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).
    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 $\mathbf{b}^{\text{vol}}_{\mathrm{elem}}$ (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 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 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 $\mathbf{b}^{\text{vol}}_{\text{elem}}$ (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 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>`

- `mBoundaryElements`

```
    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* $[N_1(\xi), \dots, N_n(\xi)] = [\phi_1(\mathbf{x}(\xi)), \dots, \phi_n(\mathbf{x}(\xi))]$

```
    GetTransformedDerivatives(xi, J)
```

▷ *similarly, returns vector with entries* $\nabla\phi_i = J\nabla_\xi N_i$

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* $[N_1(\xi), \dots, N_n(\xi)] = [\phi_1(\mathbf{x}(\xi)), \dots, \phi_n(\mathbf{x}(\xi))]$

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

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 $K_{ij} = \int_\Omega \phi_i \phi_j \,\mathrm{d}V$
    - 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()` and `basis_func.GetValues(x)` 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 $K\mathbf{U} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$ 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 $K_{ij} = \int_\Omega \phi_i \phi_j \,\mathrm{d}V$
    - 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()` and `basis_func.GetValues(x)` 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 $K\mathbf{U} = \mathbf{b}^{\text{vol}} + \mathbf{b}^{\text{surf}}$ 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 $b_i^{\mathrm{vol}} = \int_\Omega f \phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\mathrm{surf}} = \int_{\Gamma_2} g \phi_i \, \mathrm{d}S$
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 $b_i^{\mathrm{vol}} = \int_\Omega f \phi_i \,\mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\mathrm{surf}} = \int_{\Gamma_2} g \phi_i \,\mathrm{d}S$
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 $b_i^{\text{vol}} = \int_\Omega f\phi_i \,\mathrm{d}V$
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 $b_i^{\textbf{vol}} = \int_\Omega f \phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\textbf{surf}} = \int_{\Gamma_2} g \phi_i \, \mathrm{d}S$
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 $b_i^{\textsf{vol}} = \int_\Omega f \phi_i \, \mathrm{d}V$
4. Loop over Neumann boundary elements (using `boundaryConditions`) and assemble $b_i^{\textsf{surf}} = \int_{\Gamma_2} g \phi_i \, \mathrm{d}S$
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

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

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

Chaste

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

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

where

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

`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*

Chaste

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*

Chaste

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`[2] *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*

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

Chaste

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`[2] *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*

[2] 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

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

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

Chaste

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

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.

Coupled/Nonlinear PDEs

Coupled (linear) PDEs in Chaste

We consider solving a set of linear coupled PDEs, and assume it is a case in which the use of linear basis functions for all unknowns is appropriate (for example, a set of reaction-diffusion equations).

It is not possible to write a generic 'PDE class' for all such coupled systems, so a user wishing to solve such systems in Chaste will have to write their own solver.

However, using the tools available, this requires significantly less work than coding up from scratch

Chaste

Coupled (linear) PDEs in Chaste

We consider solving a set of linear coupled PDEs, and assume it is a case in which the use of linear basis functions for all unknowns is appropriate (for example, a set of reaction-diffusion equations).

It is not possible to write a generic ‘PDE class’ for all such coupled systems, so a user wishing to solve such systems in Chaste will have to write their own solver.

However, using the tools available, this requires significantly less work than coding up from scratch

Coupled (linear) PDEs in Chaste

We consider solving a set of linear coupled PDEs, and assume it is a case in which the use of linear basis functions for all unknowns is appropriate (for example, a set of reaction-diffusion equations).

It is not possible to write a generic ‘PDE class’ for all such coupled systems, so a user wishing to solve such systems in Chaste will have to write their own solver.

However, using the tools available, this requires significantly less work than coding up from scratch

The parameter PROBLEM_DIM

Define PROBLEM_DIM to be the size of the system of PDEs. For example, for the PDE system

$$\begin{aligned} u_t &= \nabla^2 u + v \\ v_t &= \nabla^2 v + a\nabla^2 u \\ w_t &= \nabla^2 w + u \end{aligned}$$

we have PROBLEM_DIM equal to 3

The PDE solver classes are written to work with general PROBLEM_DIM. In particular the following classes are all templated over this:

```
BoundaryConditionsContainer<ELEM_DIM, SPACE_DIM, PROBLEM_DIM>
AbstractFeObjectAssembler<ELEM_DIM, SPACE_DIM, PROBLEM_DIM,
                          CAN_ASSEMBLE_VEC, CAN_ASSEMBLE_MAT>
AbstractLinearPdeSolver<ELEM_DIM, SPACE_DIM, PROBLEM_DIM>
```

The parameter `PROBLEM_DIM`

Define `PROBLEM_DIM` to be the size of the system of PDEs. For example, for the PDE system

$$\begin{aligned} u_t &= \nabla^2 u + v \\ v_t &= \nabla^2 v + a\nabla^2 u \\ w_t &= \nabla^2 w + u \end{aligned}$$

we have `PROBLEM_DIM` equal to 3

The PDE solver classes are written to work with general `PROBLEM_DIM`. In particular the following classes are all templated over this:

```
BoundaryConditionsContainer<ELEM_DIM, SPACE_DIM, PROBLEM_DIM>
AbstractFeObjectAssembler<ELEM_DIM, SPACE_DIM, PROBLEM_DIM,
                          CAN_ASSEMBLE_VEC, CAN_ASSEMBLE_MAT>
AbstractLinearPdeSolver<ELEM_DIM, SPACE_DIM, PROBLEM_DIM>
```

Chaste

Striping

With the system

$$\begin{aligned} u_t &= \nabla^2 u + v \\ v_t &= \nabla^2 v + \alpha \nabla^2 u \\ w_t &= \nabla^2 w + u \end{aligned}$$

and with a mesh of N nodes, and linear basis functions for each unknown, the unknown vectors will be $\mathbf{U}^n$, $\mathbf{V}^n$, $\mathbf{W}^n$, each of size N.

In the linear system to be set up to solve this problem, the solution vector is chosen to be *striped*, i.e. the full solution vector is given by

$$\mathcal{U}^n = [U_1^n, V_1^n, W_1^n, U_2^n, V_2^n, W_2^n, \dots, U_N^n, V_N^n, W_N^n]$$

This is largely for parallelisation reasons.

The *code uses striping*, on paper however (for clarity) we use *blocks*

$$\mathcal{U}^n = [\mathbf{U}^n \ \mathbf{V}^n \ \mathbf{W}^n]$$

Striping

With the system

$$\begin{aligned} u_t &= \nabla^2 u + v \\ v_t &= \nabla^2 v + \alpha \nabla^2 u \\ w_t &= \nabla^2 w + u \end{aligned}$$

and with a mesh of N nodes, and linear basis functions for each unknown, the unknown vectors will be $\mathbf{U}^n$, $\mathbf{V}^n$, $\mathbf{W}^n$, each of size N.

In the linear system to be set up to solve this problem, the solution vector is chosen to be *striped*, i.e. the full solution vector is given by

$$\mathcal{U}^n = [U_1^n, V_1^n, W_1^n, U_2^n, V_2^n, W_2^n, \dots, U_N^n, V_N^n, W_N^n]$$

This is largely for parallelisation reasons.

The *code uses striping*, on paper however (for clarity) we use *blocks*

$$\mathcal{U}^n = [\mathbf{U}^n \ \mathbf{V}^n \ \mathbf{W}^n]$$

Striping

With the system

$$\begin{aligned} u_t &= \nabla^2 u + v \\ v_t &= \nabla^2 v + \alpha \nabla^2 u \\ w_t &= \nabla^2 w + u \end{aligned}$$

and with a mesh of N nodes, and linear basis functions for each unknown, the unknown vectors will be $\mathbf{U}^n$, $\mathbf{V}^n$, $\mathbf{W}^n$, each of size N.

In the linear system to be set up to solve this problem, the solution vector is chosen to be *striped*, i.e. the full solution vector is given by

$$\mathcal{U}^n = [U_1^n, V_1^n, W_1^n, U_2^n, V_2^n, W_2^n, \dots, U_N^n, V_N^n, W_N^n]$$

This is largely for parallelisation reasons.

The *code uses striping*, on paper however (for clarity) we use *blocks*

$$\mathcal{U}^n = [\mathbf{U}^n \ \mathbf{V}^n \ \mathbf{W}^n]$$

Weak form

Consider the static problem:

$$\begin{aligned} \nabla^2 u + \alpha \nabla^2 v + f &= 0 \\ \nabla^2 v + u + w &= 0 \\ \nabla^2 w + \beta \nabla^2 v &= 0 \end{aligned}$$

subject to $u = v = w = 0$ on Γ_1 and natural boundary conditions on Γ_2.

The linear system (in block form) can be read off to be

$$\begin{bmatrix} K & \alpha K & 0 \\ -M & K & -M \\ 0 & \beta K & K \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \\ 0 \end{bmatrix}$$

Weak form

Consider the static problem:

$$\begin{aligned} \nabla^2 u + \alpha\nabla^2 v + f &= 0 \\ \nabla^2 v + u + w &= 0 \\ \nabla^2 w + \beta\nabla^2 v &= 0 \end{aligned}$$

subject to $u = v = w = 0$ on Γ_1 and natural boundary conditions on Γ_2.

The linear system (in block form) can be read off to be

$$\begin{bmatrix} K & \alpha K & 0 \\ -M & K & -M \\ 0 & \beta K & K \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{V} \\ \mathbf{W} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \\ 0 \end{bmatrix}$$

Coupled problems in Chaste

For such coupled linear PDEs, it is reasonably straightforward to set-up a (parallel, efficient, trustworthy) solver in Chaste.

The user needs to be able to convert their set of PDEs into a linear system as above, then only needs to implement functions `ComputeMatrixTerm()` and `ComputeVectorTerm()` saying what (the elemental contributions of) the matrix and vector are (remembering the striped nature of the data structures).

For examples, see tutorial on writing PDE solvers

Nonlinear problems

Consider a nonlinear elliptic problem, such as

$$\nabla \cdot (D(u)\boldsymbol{\nabla} u) + f = 0$$

with boundary conditions

$$\begin{aligned} u &= 0 && \text{on } \Gamma_1 \\ D(u)\boldsymbol{\nabla} u \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

Computing the weak form as before, we obtain: find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega (D(u)\boldsymbol{\nabla} u) \cdot \boldsymbol{\nabla} v \, \mathrm{d}V - \int_\Omega fv \, \mathrm{d}V - \int_{\Gamma_2} gv \, \mathrm{d}S = 0 \qquad \forall v \in \mathcal{V}_0$$

Write this as: find $u \in \mathcal{V}_0$ satisfying

$$\mathcal{F}(u, v) = 0 \qquad \forall v \in \mathcal{V}_0$$

Nonlinear problems

Consider a nonlinear elliptic problem, such as

$$\nabla \cdot (D(u)\boldsymbol{\nabla} u) + f = 0$$

with boundary conditions

$$\begin{aligned} u &= 0 && \text{on } \Gamma_1 \\ D(u)\boldsymbol{\nabla} u \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

Computing the weak form as before, we obtain: find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega (D(u)\boldsymbol{\nabla} u) \cdot \boldsymbol{\nabla} v \,\mathrm{d}V - \int_\Omega f v \,\mathrm{d}V - \int_{\Gamma_2} g v \,\mathrm{d}S = 0 \qquad \forall v \in \mathcal{V}_0$$

Write this as: find $u \in \mathcal{V}_0$ satisfying

$$\mathcal{F}(u, v) = 0 \qquad \forall v \in \mathcal{V}_0$$

Nonlinear problems

Consider a nonlinear elliptic problem, such as

$$\nabla \cdot (D(u)\boldsymbol{\nabla} u) + f = 0$$

with boundary conditions

$$\begin{aligned} u &= 0 && \text{on } \Gamma_1 \\ D(u)\boldsymbol{\nabla} u \cdot \mathbf{n} &= g && \text{on } \Gamma_2 \end{aligned}$$

Computing the weak form as before, we obtain: find $u \in \mathcal{V}_0$ satisfying

$$\int_\Omega (D(u)\boldsymbol{\nabla} u) \cdot \boldsymbol{\nabla} v \,\mathrm{d}V - \int_\Omega fv \,\mathrm{d}V - \int_{\Gamma_2} gv \,\mathrm{d}S = 0 \qquad \forall v \in \mathcal{V}_0$$

Write this as: find $u \in \mathcal{V}_0$ satisfying

$$\mathcal{F}(u, v) = 0 \qquad \forall v \in \mathcal{V}_0$$

Nonlinear problems

The finite element problem is obtained as before: find $u_h \in \mathcal{V}_0^h$ satisfying

$$\mathcal{F}(u_h, v) = 0 \qquad \forall v \in \mathcal{V}_0^h$$

i.e. find coefficients $U_1, \dots, U_N$ of $u_h = \sum U_i \phi_i$ such that

$$\mathcal{F}(u_h, \phi_i) = 0 \qquad \text{for } i = 1, \dots, N$$

This is a general N-dimensional nonlinear system.

An iterative approach is required to solve nonlinear systems. Let u_h^k (equivalently, $\mathbf{U}^k = [U_1^k, \dots, U_N^k]$) be the current guess. Then the vector $\mathbf{F}^k$ defined by

$$F_i^k = \mathcal{F}(u_h^k, \phi_i)$$

is known as the k-th **residual vector**. We require a guess satisfying

$$\|\mathbf{F}^k\| < \mathrm{TOL}$$

Nonlinear problems

The finite element problem is obtained as before: find $u_h \in \mathcal{V}_0^h$ satisfying

$$\mathcal{F}(u_h, v) = 0 \qquad \forall v \in \mathcal{V}_0^h$$

i.e. find coefficients $U_1, \dots, U_N$ of $u_h = \sum U_i \phi_i$ such that

$$\mathcal{F}(u_h, \phi_i) = 0 \qquad \text{for } i = 1, \dots, N$$

This is a general N-dimensional nonlinear system.

An iterative approach is required to solve nonlinear systems. Let u_h^k (equivalently, $\mathbf{U}^k = [U_1^k, \dots, U_N^k]$) be the current guess. Then the vector $\mathbf{F}^k$ defined by

$$F_i^k = \mathcal{F}(u_h^k, \phi_i)$$

is known as the k-th **residual vector**. We require a guess satisfying

$$\|\mathbf{F}^k\| < \mathrm{TOL}$$

Nonlinear problems

The finite element problem is obtained as before: find $u_h \in \mathcal{V}_0^h$ satisfying

$$\mathcal{F}(u_h, v) = 0 \qquad \forall v \in \mathcal{V}_0^h$$

i.e. find coefficients $U_1, \dots, U_N$ of $u_h = \sum U_i \phi_i$ such that

$$\mathcal{F}(u_h, \phi_i) = 0 \qquad \text{for } i = 1, \dots, N$$

This is a general N-dimensional nonlinear system.

An iterative approach is required to solve nonlinear systems. Let u_h^k (equivalently, $\mathbf{U^k} = [U_1^k, \dots, U_N^k]$) be the current guess. Then the vector $\mathbf{F}^k$ defined by

$$F_i^k = \mathcal{F}(u_h^k, \phi_i)$$

is known as the k-th **residual vector**. We require a guess satisfying

$$\|\mathbf{F}^k\| < \mathrm{TOL}$$

Newton's method

Suppose we want to solve the nonlinear set of N equations

$$\mathbf{F}(\mathbf{U}) = 0$$

Given an initial guess U^0, Newton's method is: let $\mathbf{U}^{k+1} = \mathbf{U}^k + \delta\mathbf{U}^{k+1}$, where $\delta\mathbf{U}^{k+1}$ satisfies the linear system

$$J(\mathbf{U}^k)\,\delta\mathbf{U}^{k+1} = -\mathbf{F}(\mathbf{U}^k)$$

where $J_{ij} = \frac{\partial F_i}{\partial U_j}$.

Newton's method provides quadratic convergence when the current guess is 'close enough' to the true solution. To avoid initial divergence however, it may be necessary to use **damping**

$$\mathbf{U}^{k+1} = \mathbf{U}^k + s^k \delta\mathbf{U}^{k+1}$$

for some s^k generally smaller than 1. (There are various ways to go about choosing s^k, the simplest is to pick one from a small list of possibilities which leads to the smallest $\|\mathbf{F}\|$).

Newton's method

Suppose we want to solve the nonlinear set of N equations

$$\mathbf{F}(\mathbf{U}) = 0$$

Given an initial guess U^0, Newton's method is: let $\mathbf{U}^{k+1} = \mathbf{U}^k + \delta\mathbf{U}^{k+1}$, where $\delta\mathbf{U}^{k+1}$ satisfies the linear system

$$J(\mathbf{U}^k)\,\delta\mathbf{U}^{k+1} = -\mathbf{F}(\mathbf{U}^k)$$

where $J_{ij} = \frac{\partial F_i}{\partial U_j}$.

Newton's method provides quadratic convergence when the current guess is 'close enough' to the true solution. To avoid initial divergence however, it may be necessary to use **damping**

$$\mathbf{U}^{k+1} = \mathbf{U}^k + s^k\delta\mathbf{U}^{k+1}$$

for some s^k generally smaller than 1. (There are various ways to go about choosing s^k, the simplest is to pick one from a small list of possibilities which leads to the smallest $\|\mathbf{F}\|$).

Alternative nonlinear solvers

- There are other methods for solving nonlinear systems, for example solve $x = f(x)$ using **fixed point iterations**: $x^{n+1} = f(x^n)$.
- For $\mathbf{F}(\mathbf{U}) = 0$, this is

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \mathbf{F}(\mathbf{U}^n)$$

- Very loosely speaking, methods which use the Jacobian will be more effective.
- If used, the Jacobian can be either provided analytically (if so, has to be calculated on paper on paper and coded up); or estimated numerically (slow).

- Petsc has (black-box) solvers for nonlinear systems. The user has to provide functions telling Petsc how to compute the residual (and optionally, the Jacobian)
- Chaste sometimes uses the Petsc nonlinear solvers (eg `AbstractNonlinearAssemblerSolverHybrid`), sometimes Newton's method is coded from scratch (solid mechanics solvers).

Alternative nonlinear solvers

- There are other methods for solving nonlinear systems, for example solve $x = f(x)$ using **fixed point iterations**: $x^{n+1} = f(x^n)$.
- For $\mathbf{F}(\mathbf{U}) = 0$, this is
$$\mathbf{U}^{n+1} = \mathbf{U}^n + \mathbf{F}(\mathbf{U}^n)$$
- Very loosely speaking, methods which use the Jacobian will be more effective.
- If used, the Jacobian can be either provided analytically (if so, has to be calculated on paper on paper and coded up); or estimated numerically (slow).
- Petsc has (black-box) solvers for nonlinear systems. The user has to provide functions telling Petsc how to compute the residual (and optionally, the Jacobian)
- Chaste sometimes uses the Petsc nonlinear solvers (eg `AbstractNonlinearAssemblerSolverHybrid`), sometimes Newton's method is coded from scratch (solid mechanics solvers).

Alternative nonlinear solvers

- There are other methods for solving nonlinear systems, for example solve $x = f(x)$ using **fixed point iterations**: $x^{n+1} = f(x^n)$.
- For $\mathbf{F}(\mathbf{U}) = 0$, this is
$$\mathbf{U}^{n+1} = \mathbf{U}^n + \mathbf{F}(\mathbf{U}^n)$$
- Very loosely speaking, methods which use the Jacobian will be more effective.
- If used, the Jacobian can be either provided analytically (if so, has to be calculated on paper on paper and coded up); or estimated numerically (slow).
- Petsc has (black-box) solvers for nonlinear systems. The user has to provide functions telling Petsc how to compute the residual (and optionally, the Jacobian)
- Chaste sometimes uses the Petsc nonlinear solvers (eg `AbstractNonlinearAssemblerSolverHybrid`), sometimes Newton's method is coded from scratch (solid mechanics solvers).

Alternative nonlinear solvers

- There are other methods for solving nonlinear systems, for example solve $x = f(x)$ using **fixed point iterations**: $x^{n+1} = f(x^n)$.
- For $\mathbf{F}(\mathbf{U}) = 0$, this is

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \mathbf{F}(\mathbf{U}^n)$$

- Very loosely speaking, methods which use the Jacobian will be more effective.
- If used, the Jacobian can be either provided analytically (if so, has to be calculated on paper on paper and coded up); or estimated numerically (slow).

- Petsc has (black-box) solvers for nonlinear systems. The user has to provide functions telling Petsc how to compute the residual (and optionally, the Jacobian)
- Chaste sometimes uses the Petsc nonlinear solvers (eg `AbstractNonlinearAssemblerSolverHybrid`), sometimes Newton's method is coded from scratch (solid mechanics solvers).

Alternative nonlinear solvers

- There are other methods for solving nonlinear systems, for example solve $x = f(x)$ using **fixed point iterations**: $x^{n+1} = f(x^n)$.
- For $\mathbf{F}(\mathbf{U}) = 0$, this is
$$\mathbf{U}^{n+1} = \mathbf{U}^n + \mathbf{F}(\mathbf{U}^n)$$

- Very loosely speaking, methods which use the Jacobian will be more effective.
- If used, the Jacobian can be either provided analytically (if so, has to be calculated on paper on paper and coded up); or estimated numerically (slow).

- Petsc has (black-box) solvers for nonlinear systems. The user has to provide functions telling Petsc how to compute the residual (and optionally, the Jacobian)
- Chaste sometimes uses the Petsc nonlinear solvers (eg `AbstractNonlinearAssemblerSolverHybrid`), sometimes Newton's method is coded from scratch (solid mechanics solvers).

Solving nonlinear problems with finite elements and Newton's method

Firstly, decide whether the Jacobian (if used) is to be computed numerically or analytically. If the latter, both the residual and Jacobian need to be 'assembled' in a finite element manner.

- Choose a (good!) initial guess $\mathbf{U}^0$
- Compute the initial residual $\mathbf{F}^0 = \mathbf{F}(\mathbf{U}^0)$ (loop over elements, compute elemental contribution, add to full vector).
- While $\|\mathbf{F}^k\| > \mathrm{TOL}$
    - Compute $J(\mathbf{U}^k)$ (loop over elements, compute elemental contribution, add to full matrix)
    - Solve $J\delta\mathbf{U} = -\mathbf{F}^k$
    - Set $\mathbf{U}^{k+1} = \mathbf{U}^k + s\,\delta\mathbf{U}$, choosing s appropriately
    - Compute $\mathbf{F}^{k+1}$ (loop over elements, compute elemental contribution, add to full vector). [Now increment k]

Solving nonlinear problems with finite elements and Newton's method

Firstly, decide whether the Jacobian (if used) is to be computed numerically or analytically. If the latter, both the residual and Jacobian need to be 'assembled' in a finite element manner.

- Choose a (good!) initial guess $\mathbf{U}^0$
- Compute the initial residual $\mathbf{F}^0 = \mathbf{F}(\mathbf{U}^0)$ (loop over elements, compute elemental contribution, add to full vector).
- While $\|\mathbf{F}^k\| > \mathrm{TOL}$
    - Compute $J(\mathbf{U}^k)$ (loop over elements, compute elemental contribution, add to full matrix).
    - Solve $J\delta\mathbf{U} = -\mathbf{F}^k$
    - Set $\mathbf{U}^{k+1} = \mathbf{U}^k + s\delta\mathbf{U}$, choosing s appropriately
    - Compute $\mathbf{F}^{k+1}$ (loop over elements, compute elemental contribution, add to full vector). [Now increment k]

Solving nonlinear problems with finite elements and Newton's method

Firstly, decide whether the Jacobian (if used) is to be computed numerically or analytically. If the latter, both the residual and Jacobian need to be 'assembled' in a finite element manner.

- Choose a (good!) initial guess $\mathbf{U}^0$
- Compute the initial residual $\mathbf{F}^0 = \mathbf{F}(\mathbf{U}^0)$ (loop over elements, compute elemental contribution, add to full vector).
- While $\|\mathbf{F}^k\| > \mathrm{TOL}$
  - Compute $J(\mathbf{U}^k)$ (loop over elements, compute elemental contribution, add to full matrix).
  - Solve $J\,\delta\mathbf{U} = -\mathbf{F}^k$.
  - Set $\mathbf{U}^{k+1} = \mathbf{U}^k + s\,\delta\mathbf{U}$, choosing s appropriately
  - Compute $\mathbf{F}^{k+1}$ (loop over elements, compute elemental contribution, add to full vector). [Now increment k]

Solving nonlinear problems with finite elements and Newton's method

Firstly, decide whether the Jacobian (if used) is to be computed numerically or analytically. If the latter, both the residual and Jacobian need to be 'assembled' in a finite element manner.

- Choose a (good!) initial guess $\mathbf{U}^0$
- Compute the initial residual $\mathbf{F}^0 = \mathbf{F}(\mathbf{U}^0)$ (loop over elements, compute elemental contribution, add to full vector).
- While $\|\mathbf{F}^k\| > \mathrm{TOL}$
  - Compute $J(\mathbf{U}^k)$ (loop over elements, compute elemental contribution, add to full matrix).
  - Solve $J\,\delta\mathbf{U} = -\mathbf{F}^k$.
  - Set $\mathbf{U}^{k+1} = \mathbf{U}^k + s\,\delta\mathbf{U}$, choosing s appropriately.
  - Compute $\mathbf{F}^{k+1}$ (loop over elements, compute elemental contribution, add to full vector). [Now increment k]

Solving nonlinear problems with finite elements and Newton's method

Firstly, decide whether the Jacobian (if used) is to be computed numerically or analytically. If the latter, both the residual and Jacobian need to be 'assembled' in a finite element manner.

- Choose a (good!) initial guess $\mathbf{U}^0$
- Compute the initial residual $\mathbf{F}^0 = \mathbf{F}(\mathbf{U}^0)$ (loop over elements, compute elemental contribution, add to full vector).
- While $\|\mathbf{F}^k\| > \mathrm{TOL}$
  - Compute $J(\mathbf{U}^k)$ (loop over elements, compute elemental contribution, add to full matrix).
  - Solve $J\,\delta\mathbf{U} = -\mathbf{F}^k$.
  - Set $\mathbf{U}^{k+1} = \mathbf{U}^k + s\,\delta\mathbf{U}$, choosing s appropriately.
  - Compute $\mathbf{F}^{k+1}$ (loop over elements, compute elemental contribution, add to full vector). [Now increment k]

Solving nonlinear problems with finite elements and Newton's method

Firstly, decide whether the Jacobian (if used) is to be computed numerically or analytically. If the latter, both the residual and Jacobian need to be 'assembled' in a finite element manner.

- Choose a (good!) initial guess $\mathbf{U}^0$
- Compute the initial residual $\mathbf{F}^0 = \mathbf{F}(\mathbf{U}^0)$ (loop over elements, compute elemental contribution, add to full vector).
- While $\|\mathbf{F}^k\| > \mathrm{TOL}$
    - Compute $J(\mathbf{U}^k)$ (loop over elements, compute elemental contribution, add to full matrix).
    - Solve $J\,\delta\mathbf{U} = -\mathbf{F}^k$.
    - Set $\mathbf{U}^{k+1} = \mathbf{U}^k + s\,\delta\mathbf{U}$, choosing s appropriately.
    - Compute $\mathbf{F}^{k+1}$ (loop over elements, compute elemental contribution, add to full vector). [Now increment k]

Solving nonlinear problems with finite elements and Newton's method

Firstly, decide whether the Jacobian (if used) is to be computed numerically or analytically. If the latter, both the residual and Jacobian need to be 'assembled' in a finite element manner.

- Choose a (good!) initial guess $\mathbf{U}^0$
- Compute the initial residual $\mathbf{F}^0 = \mathbf{F}(\mathbf{U}^0)$ (loop over elements, compute elemental contribution, add to full vector).
- While $\|\mathbf{F}^k\| > \mathrm{TOL}$
  - Compute $J(\mathbf{U}^k)$ (loop over elements, compute elemental contribution, add to full matrix).
  - Solve $J\,\delta\mathbf{U} = -\mathbf{F}^k$.
  - Set $\mathbf{U}^{k+1} = \mathbf{U}^k + s\,\delta\mathbf{U}$, choosing s appropriately.
  - Compute $\mathbf{F}^{k+1}$ (loop over elements, compute elemental contribution, add to full vector). [Now increment k]

Cardiac electro-physiology

The monodomain and bidomain equations

The monodomain equations (dropping stimulus currents) is essentially the heat equation coupled to ODEs:

$$\begin{aligned}\chi\left(\mathcal{C}\frac{\partial V}{\partial t}+I_{\text{ion}}(\mathbf{u},V)\right)-\nabla\cdot(\sigma\nabla V) &= 0\\ \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}\left(\mathbf{u},V\right)\end{aligned}$$

(with zero-Neumann BCs on entire boundary)

The bidomain equations can be written as a parabolic PDE coupled to an elliptic PDE coupled to ODEs:

$$\begin{aligned}\chi\left(\mathcal{C}\frac{\partial V}{\partial t}+I_{\text{ion}}(\mathbf{u},V)\right)-\nabla\cdot\left(\sigma_i\nabla\left(V+\phi_e\right)\right) &= 0\\ \nabla\cdot\left(\sigma_i\nabla V+\left(\sigma_i+\sigma_e\right)\nabla\phi_e\right) &= 0\\ \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}\left(\mathbf{u},V\right)\end{aligned}$$

(with zero-Neumann BCs on entire boundary)

The monodomain and bidomain equations

The monodomain equations (dropping stimulus currents) is essentially the heat equation coupled to ODEs:

$$\begin{aligned} \chi\left(\mathcal{C}\frac{\partial V}{\partial t} + I_{\text{ion}}(\mathbf{u}, V)\right) - \nabla\cdot(\sigma\boldsymbol{\nabla} V) &= 0 \\ \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}\left(\mathbf{u}, V\right) \end{aligned}$$

(with zero-Neumann BCs on entire boundary)

The bidomain equations can be written as a parabolic PDE coupled to an elliptic PDE coupled to ODEs:

$$\begin{aligned} \chi\left(\mathcal{C}\frac{\partial V}{\partial t} + I_{\text{ion}}(\mathbf{u}, V)\right) - \nabla\cdot(\sigma_i\boldsymbol{\nabla}\left(V + \phi_e\right)) &= 0 \\ \nabla\cdot\left(\sigma_i\boldsymbol{\nabla} V + \left(\sigma_i + \sigma_e\right)\boldsymbol{\nabla}\phi_e\right) &= 0 \\ \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}\left(\mathbf{u}, V\right) \end{aligned}$$

(with zero-Neumann BCs on entire boundary)

Placing cell models

We know that the forcing term of the heat equation enters the RHS vector of the FEM discretisation as (using ψ rather than ϕ for basis functions):

$$b_j = \int_\Omega f \psi_j \, \mathrm{d}V$$

which here is (assuming the reaction term is treated explicitly in the time-discretisation)

$$b_j^{n+1} = \int_\Omega I_{\text{ion}}(\mathbf{u}^n, V^n) \psi_j \, \mathrm{d}V$$

Therefore we require the ionic current at the quadrature points, i.e. $\mathbf{u}$ *is required at the quadrature points*. The natural approach is therefore to solve cell models at quad points.

However this can be computationally-expensive (and a pain to implement), instead, solve cell models at nodes and interpolate onto quadrature points.

Placing cell models

We know that the forcing term of the heat equation enters the RHS vector of the FEM discretisation as (using ψ rather than ϕ for basis functions):

$$b_j = \int_\Omega f \psi_j \,\mathrm{d}V$$

which here is (assuming the reaction term is treated explicitly in the time-discretisation)

$$b_j^{n+1} = \int_\Omega I_{\text{ion}}(\mathbf{u}^n, V^n)\psi_j \,\mathrm{d}V$$

Therefore we require the ionic current at the quadrature points, i.e. $\mathbf{u}$ *is required at the quadrature points.* The natural approach is therefore to solve cell models at quad points.

However this can be computationally-expensive (and a pain to implement), instead, solve cell models at nodes and interpolate onto quadrature points.

Placing cell models

We know that the forcing term of the heat equation enters the RHS vector of the FEM discretisation as (using ψ rather than ϕ for basis functions):

$$b_j = \int_\Omega f \psi_j \, \mathrm{d}V$$

which here is (assuming the reaction term is treated explicitly in the time-discretisation)

$$b_j^{n+1} = \int_\Omega I_{\text{ion}}(\mathbf{u}^n, V^n)\psi_j \, \mathrm{d}V$$

Therefore we require the ionic current at the quadrature points, i.e. **u** *is required at the quadrature points*. The natural approach is therefore to solve cell models at quad points.

However this can be computationally-expensive (and a pain to implement), instead, solve cell models at nodes and interpolate onto quadrature points.

Placing cell models

We know that the forcing term of the heat equation enters the RHS vector of the FEM discretisation as (using ψ rather than ϕ for basis functions):

$$b_j = \int_\Omega f \psi_j \,\mathrm{d}V$$

which here is (assuming the reaction term is treated explicitly in the time-discretisation)

$$b_j^{n+1} = \int_\Omega I_{\text{ion}}(\mathbf{u}^n, V^n)\psi_j \,\mathrm{d}V$$

Therefore we require the ionic current at the quadrature points, i.e. **u** *is required at the quadrature points*. The natural approach is therefore to solve cell models at quad points.

However this can be computationally-expensive (and a pain to implement), instead, solve cell models at nodes and interpolate onto quadrature points.

Cell models at nodes

Solving cell models at nodes, we write the ionic current evaluated at the nodes as $\mathbf{I} = (I_1, \ldots, I_N)$. Interpolating the ionic current onto the quadrature point using linear basis functions ψ_j, we have

$$I_{\text{ion}} = \sum I_k \psi_k$$

which means that

$$b_j = \int_\Omega I_{\text{ion}} \psi_j \,\mathrm{d}V = \int_\Omega \sum_k I_k \psi_k \psi_j \,\mathrm{d}V = \sum_k I_k \int_\Omega \psi_k \psi_j \,\mathrm{d}V = \sum_k M_{jk} I_k$$

so that

$$\mathbf{b} = M\mathbf{I}$$

Monodomain discretisation

Solve

$$\begin{aligned} \chi\left(\mathcal{C}\frac{\partial V}{\partial t} + I_{\text{ion}}(\mathbf{u}, V)\right) - \nabla\cdot(\sigma\boldsymbol{\nabla} V) &= 0 \\ \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}\,(\mathbf{u}, V) \end{aligned}$$

subject to initial conditions and zero-Neumann boundary conditions.

We de-couple the ODEs from the PDEs, and use a time-discretisation which treats the conductivity implicitly and the (nonlinear) reaction term explicitly, and place cell models at nodes, obtaining, for the PDE solve:

$$\left(\frac{\chi\mathcal{C}}{\Delta t}M + K\right)\mathbf{V}^{m+1} = \frac{\chi\mathcal{C}}{\Delta t}M\mathbf{V}^{m} - M\mathbf{I}^{m}$$

Monodomain discretisation

Solve

$$\begin{aligned}\chi\left(\mathcal{C}\frac{\partial V}{\partial t} + I_{\text{ion}}(\mathbf{u}, V)\right) - \nabla\cdot(\sigma\boldsymbol{\nabla} V) &= 0\\ \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}(\mathbf{u}, V)\end{aligned}$$

subject to initial conditions and zero-Neumann boundary conditions.

We de-couple the ODEs from the PDEs, and use a time-discretisation which treats the conductivity implicitly and the (nonlinear) reaction term explicitly, and place cell models at nodes, obtaining, for the PDE solve:

$$\left(\frac{\chi\mathcal{C}}{\Delta t}M + K\right)\mathbf{V}^{m+1} = \frac{\chi\mathcal{C}}{\Delta t}M\mathbf{V}^{m} - M\mathbf{I}^{m}$$

Chaste

Bidomain discretisation

Solve

$$
\begin{aligned}
\chi\left(\mathcal{C}\frac{\partial V}{\partial t} + I_{\text{ion}}(\mathbf{u}, V)\right) - \nabla\cdot\left(\sigma_i \boldsymbol{\nabla}\left(V + \phi_e\right)\right) &= 0\\
\nabla\cdot\left(\sigma_i \boldsymbol{\nabla} V + \left(\sigma_i + \sigma_e\right)\boldsymbol{\nabla}\phi_e\right) &= 0\\
\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}\left(\mathbf{u}, V\right)
\end{aligned}
$$

subject to initial conditions and zero-Neumann boundary conditions.

Going through the same procedure, we obtain

$$
\begin{bmatrix} \frac{\chi\mathcal{C}}{\Delta t}M + K[\sigma_i] & K[\sigma_i] \\ K[\sigma_i] & K[\sigma_i + \sigma_e] \end{bmatrix}
\begin{bmatrix} \mathbf{V}^{m+1} \\ \boldsymbol{\Phi}_e^{m+1} \end{bmatrix}
=
\begin{bmatrix} \frac{\chi\mathcal{C}}{\Delta t}M\mathbf{V}^m - M\mathbf{I}^m \\ 0 \end{bmatrix}
$$

where

$$
K[\sigma]_{jk} = \int_{\Omega} \boldsymbol{\nabla}\psi_k \cdot \left(\sigma\boldsymbol{\nabla}\psi_j\right)\mathrm{d}V
$$

Bidomain discretisation

Solve

$$\begin{aligned}
\chi\left(\mathcal{C}\frac{\partial V}{\partial t} + I_{\text{ion}}(\mathbf{u}, V)\right) - \nabla\cdot\left(\sigma_i \boldsymbol{\nabla}\left(V + \phi_e\right)\right) &= 0 \\
\nabla\cdot\left(\sigma_i \boldsymbol{\nabla} V + \left(\sigma_i + \sigma_e\right)\boldsymbol{\nabla}\phi_e\right) &= 0 \\
\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} &= \mathbf{f}\left(\mathbf{u}, V\right)
\end{aligned}$$

subject to initial conditions and zero-Neumann boundary conditions.

Going through the same procedure, we obtain

$$\begin{bmatrix} \frac{\chi\mathcal{C}}{\Delta t}M + K[\sigma_i] & K[\sigma_i] \\ K[\sigma_i] & K[\sigma_i + \sigma_e] \end{bmatrix} \begin{bmatrix} \mathbf{V}^{m+1} \\ \mathbf{\Phi}_e^{m+1} \end{bmatrix} = \begin{bmatrix} \frac{\chi\mathcal{C}}{\Delta t}M\mathbf{V}^m - M\mathbf{I}^m \\ 0 \end{bmatrix}$$

where

$$K[\sigma]_{jk} = \int_\Omega \boldsymbol{\nabla}\psi_k \cdot \left(\sigma\boldsymbol{\nabla}\psi_j\right)\mathrm{d}V$$

Solution overview

For either the monodomain or bidomain equations

- Set up the left-hand side matrix, A say (loop over elements, etc)
- Set up the mass matrix (loop over elements, etc)
- Set up the initial conditions $\mathbf{V}^0$, also initialise cell models at each node
- While $t < t_{\text{end}}$
  - Pass nodal voltages to each cell model
  - Solve cell models at each node using choice of ODE solver
  - Compute ionic current at each node
  - Set up linear system RHS vector (matrix-vector products only, no need for assembly)
  - Solve linear system

Cell models in Chaste

`AbstractOdeSystem`
- `mStateVariables`
- `EvaluateYDerivatives(t,y)`

`AbstractCardiacCell`[3] *inherits from* `AbstractOdeSystem` :
- `mOdeSolver`
  - *of type* *`AbstractOdeSolver`*
- `Compute(t0,t1)`
  - *Use solver to solve between given times, updating internal state*
  - *Class has various other cardiac cell related functionality*

`LuoRudyCellModel` *inherits from* `AbstractCardiacCell`:
- *Implemented method:* `EvaluateYDerivatives(t,y)`

[3] Slightly simplified

Cell models in Chaste

```
AbstractOdeSystem
    mStateVariables
    EvaluateYDerivatives(t,y)
```

`AbstractCardiacCell`[3] *inherits from* `AbstractOdeSystem` :

`mOdeSolver`

- ▷ *of type* `AbstractOdeSolver`

`Compute(t0,t1)`

- ▷ *Use solver to solve between given times, updating internal state*
- ▷ *Class has various other cardiac cell related functionality*

`LuoRudyCellModel` *inherits from* `AbstractCardiacCell`:

Implemented method: `EvaluateYDerivatives(t,y)`

[3]Slightly simplified

Cell models in Chaste

AbstractOdeSystem
    mStateVariables
    EvaluateYDerivatives(t,y)

AbstractCardiacCell[3] *inherits from* AbstractOdeSystem :
    mOdeSolver
        ▷ *of type* *AbstractOdeSolver*
    Compute(t0,t1)
        ▷ *Use solver to solve between given times, updating internal state*
        ▷ *Class has various other cardiac cell related functionality*

LuoRudyCellModel *inherits from* AbstractCardiacCell:
    Implemented method: EvaluateYDerivatives(t,y)

[3] Slightly simplified

Cardiac PDE solvers in Chaste

`AbstractLinearPdeSolver`:

`PrepareForSetupLinearSystem()`

- ▷ *Empty implementation here (ie does nothing)*

`SetupLinearSystem()`

`AbstractDynamicPdeSolver` *inherits from* `AbstractLinearPdeSolver` :

`SetTimes(t0,t1)`

`SetInitialCondition(initialCondition)`

`Solve()`

- ▷ *Calls `PrepareForSetupLinearSystem()`, then calls `SetupLinearSystem()`, then solves linear system.*

Chaste

Cardiac PDE solvers in Chaste

`MonodomainSolver`

- `mMonodomainTissue`
  - ▷ *Basically a set of `AbstractCardiacCell`s for each node*
  - ▷ *plus conductivity information*
- `mMonodomainAssembler`
- `mMassMatrixAssembler`
- `PrepareForSetupLinearSystem()`
  - ▷ *Overloaded to solve all the cell models*
- *Implemented method:* `SetUpLinearSystem()`
  - ▷ *Uses the above assemblers to set up the linear system*

Notes:

- `BidomainSolver` uses the same design (but uses `PROBLEM_DIM=2`)
- There is `MonodomainProblem` and `BidomainProblem` (both inheriting from `AbstractCardiacProblem`). These **own** solvers and deal with set-up and output etc.

Cardiac PDE solvers in Chaste

`MonodomainSolver`

- `mMonodomainTissue`
  - ▷ *Basically a set of `AbstractCardiacCell`s for each node*
  - ▷ *plus conductivity information*
- `mMonodomainAssembler`
- `mMassMatrixAssembler`
- `PrepareForSetupLinearSystem()`
  - ▷ *Overloaded to solve all the cell models*
- *Implemented method:* `SetUpLinearSystem()`
  - ▷ *Uses the above assemblers to set up the linear system*

Notes:

- `BidomainSolver` uses the same design (but uses `PROBLEM_DIM=2`)
- There is `MonodomainProblem` and `BidomainProblem` (both inheriting from `AbstractCardiacProblem`). These **own** solvers and deal with set-up and output etc.

Coupled reaction-diffusion equations

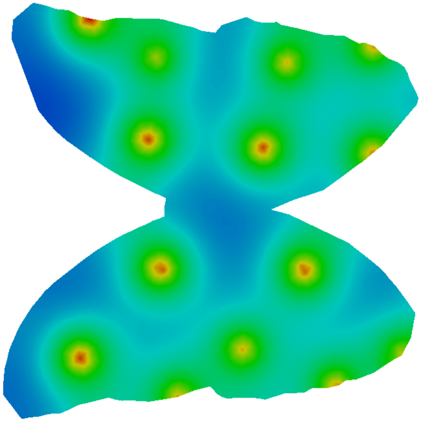
A solver for general coupled reaction-diffusion equations has recently been written.

$$\frac{\partial u_i}{\partial t} - \nabla \cdot (D_i \nabla u_i) = f_i(\mathbf{x}, u_1, \dots, u_p, v_1, \dots, v_p),$$
$$\frac{\mathrm{d}v_i}{\mathrm{d}t} = g_i(\mathbf{x}, u_1, \dots, u_p, v_1, \dots, v_p),$$

where u_i and v_i denote the extracellular and intracellular concentrations of solute i respectively, and with BCs

$$\begin{aligned} u_i &= u_i^*(\mathbf{x}), && \text{on } \Gamma_1 \\ \mathbf{n} \cdot (D_i(\mathbf{x}) \nabla u_i) &= g_i(\mathbf{x}), && \text{on } \Gamma_2 \\ u_i(0, \mathbf{x}) &= u_i^{(0)}(\mathbf{x}), \\ v_i(0, \mathbf{x}) &= v_i^{(0)}(\mathbf{x}), \end{aligned}$$

Coupled reaction-diffusion equations



See the tutorial `LinearParabolicPdeSystemsWithCoupledOdeSystems` for more details

An overview of alternative methods for solving PDEs

The finite difference method

- Finite differences are conceptually the simplest method for solving PDEs
- Set up a (generally regular) grid on the geometry, aim to compute the solution at the gridpoints
- All derivatives in the PDE (and in Neumann BCs) are replaced with difference formulas

For a regular grid in 1D $x_0, x_1, \dots, x_N$, stepsize h: some possible difference formulas and corresponding error introduced are: forward, backward and central differences:

$$\begin{aligned}
\frac{\mathrm{d}u}{\mathrm{d}x}(x_i) &= \frac{x_{i+1} - x_i}{h} + \mathcal{O}(h) \\
\frac{\mathrm{d}u}{\mathrm{d}x}(x_i) &= \frac{x_i - x_{i-1}}{h} + \mathcal{O}(h) \\
\frac{\mathrm{d}u}{\mathrm{d}x}(x_i) &= \frac{x_{i+1} - x_{i-1}}{2h} + \mathcal{O}(h^2)
\end{aligned}$$

and

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_i) = \frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} + \mathcal{O}(h^2)$$

The finite difference method

- Finite differences are conceptually the simplest method for solving PDEs
- Set up a (generally regular) grid on the geometry, aim to compute the solution at the gridpoints
- All derivatives in the PDE (and in Neumann BCs) are replaced with difference formulas

For a regular grid in 1D $x_0, x_1, \dots, x_N$, stepsize h: some possible difference formulas and corresponding error introduced are: forward, backward and central differences:

$$\begin{aligned}
\frac{\mathrm{d}u}{\mathrm{d}x}(x_i) &= \frac{x_{i+1} - x_i}{h} + \mathcal{O}(h) \\
\frac{\mathrm{d}u}{\mathrm{d}x}(x_i) &= \frac{x_i - x_{i-1}}{h} + \mathcal{O}(h) \\
\frac{\mathrm{d}u}{\mathrm{d}x}(x_i) &= \frac{x_{i+1} - x_{i-1}}{2h} + \mathcal{O}(h^2)
\end{aligned}$$

and

$$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}(x_i) = \frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} + \mathcal{O}(h^2)$$

The finite difference method

For example, for the heat equation $u_t = u_{xx} + f$, suppose we choose a fully explicit time-discretisation, and then discretise in space:

Heat equation: $$u_t = u_{xx} + f$$

Semi-discretised: $$u^{n+1} - u^n = \Delta t\, u^n_{xx} + \Delta t\, f(t_n)$$

Fully discretised: $$u^{n+1}_i - u^n_i = \frac{\Delta t}{h^2}\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right) + \Delta t\, f(t_n, x_i)$$

(Since this is an explicit scheme there will be a condition required for stability: $\frac{\Delta t}{h^2} \leq \frac{1}{2}$. Such results are obtained using *Von Neumann (Fourier) stability analysis*).

Let us write the above as a linear system: $\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t\, \mathbf{F}^n$, where $U^n = [U^n_1, \ldots, U^n_N]$, $F^n = [f(t_n, x_1), \ldots, f(t_n, x_N)]$, and D is a matrix with -2s on the diagonal and 1s above and below the diagonal.

The first and last rows of the linear system have to be altered to take into account Dirichlet and/or **Neumann** boundary conditions.

The finite difference method

For example, for the heat equation $u_t = u_{xx} + f$, suppose we choose a fully explicit time-discretisation, and then discretise in space:

Heat equation:	$u_t = u_{xx} + f$
Semi-discretised:	$u^{n+1} - u^n = \Delta t\, u^n_{xx} + \Delta t\, f(t_n)$
Fully discretised:	$u^{n+1}_i - u^n_i = \frac{\Delta t}{h^2}(u^n_{i+1} - 2u^n_i + u^n_{i-1}) + \Delta t\, f(t_n, x_i)$

(Since this is an explicit scheme there will be a condition required for stability: $\frac{\Delta t}{h^2} \leq \frac{1}{2}$. Such results are obtained using *Von Neumann (Fourier) stability analysis*).

Let us write the above as a linear system: $\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t\, \mathbf{F}^n$, where $U^n = [U^n_1, \ldots, U^n_N]$, $F^n = [f(t_n, x_1), \ldots, f(t_n, x_N)]$, and D is a matrix with -2s on the diagonal and 1s above and below the diagonal.

The first and last rows of the linear system have to be altered to take into account Dirichlet and/or **Neumann** boundary conditions.

The finite difference method

For example, for the heat equation $u_t = u_{xx} + f$, suppose we choose a fully explicit time-discretisation, and then discretise in space:

Heat equation:	$u_t = u_{xx} + f$
Semi-discretised:	$u^{n+1} - u^n = \Delta t \, u^n_{xx} + \Delta t \, f(t_n)$
Fully discretised:	$u_i^{n+1} - u_i^n = \frac{\Delta t}{h^2}(u^n_{i+1} - 2u^n_i + u^n_{i-1}) + \Delta t \, f(t_n, x_i)$

(Since this is an explicit scheme there will be a condition required for stability: $\frac{\Delta t}{h^2} \leq \frac{1}{2}$. Such results are obtained using *Von Neumann (Fourier) stability analysis*).

Let us write the above as a linear system: $\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t \, \mathbf{F}^n$, where $U^n = [U^n_1, \ldots, U^n_N]$, $F^n = [f(t_n, x_1), \ldots, f(t_n, x_N)]$, and D is a matrix with -2s on the diagonal and 1s above and below the diagonal.

The first and last rows of the linear system have to be altered to take into account Dirichlet and/or **Neumann** boundary conditions.

The finite difference method

For example, for the heat equation $u_t = u_{xx} + f$, suppose we choose a fully explicit time-discretisation, and then discretise in space:

$$\begin{aligned}
&\text{Heat equation:} && u_t = u_{xx} + f \\
&\text{Semi-discretised:} && u^{n+1} - u^n = \Delta t\, u^n_{xx} + \Delta t\, f(t_n) \\
&\text{Fully discretised:} && u_i^{n+1} - u_i^n = \frac{\Delta t}{h^2}(u^n_{i+1} - 2u^n_i + u^n_{i-1}) + \Delta t\, f(t_n, x_i)
\end{aligned}$$

(Since this is an explicit scheme there will be a condition required for stability: $\frac{\Delta t}{h^2} \leq \frac{1}{2}$. Such results are obtained using *Von Neumann (Fourier) stability analysis*).

Let us write the above as a linear system: $\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t\, \mathbf{F}^n$, where $U^n = [U_1^n, \dots, U_N^n]$, $F^n = [f(t_n, x_1), \dots, f(t_n, x_N)]$, and D is a matrix with -2s on the diagonal and 1s above and below the diagonal.

The first and last rows of the linear system have to be altered to take into account Dirichlet and/or **Neumann** boundary conditions.

The finite difference method

Compare the finite difference equation:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t\mathbf{F}^n$$

with the equivalent finite element equation:

$$M\mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, K\mathbf{U}^n + \Delta t\mathbf{b}^n$$

In fact, for a regular grid in 1D and with linear basis functions, $K = D/h$ (except for first/last rows).

The big advantages of FE over FD are

- FD is difficult to write down on **irregular geometries**, but FE works ON any valid mesh
- Neumann boundary conditions are handled very naturally in FE— require an integral over surface, and nothing required for zero-Neumann BCs. Much more difficult in FD
- Error analysis

The finite difference method

Compare the finite difference equation:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t \mathbf{F}^n$$

with the equivalent finite element equation:

$$M\mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, K\mathbf{U}^n + \Delta t \mathbf{b}^n$$

In fact, for a regular grid in 1D and with linear basis functions, $K = D/h$ (except for first/last rows).

The big advantages of FE over FD are

- FD is difficult to write down on **irregular geometries**, but FE works ON any valid mesh
- **Neumann boundary conditions** are handled very naturally in FE—require an integral over surface, and nothing required for zero-Neumann BCs. Much more difficult in FD
- Error analysis

The finite difference method

Compare the finite difference equation:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t\mathbf{F}^n$$

with the equivalent finite element equation:

$$M\mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, K\mathbf{U}^n + \Delta t\mathbf{b}^n$$

In fact, for a regular grid in 1D and with linear basis functions, $K = D/h$ (except for first/last rows).

The big advantages of FE over FD are

- FD is difficult to write down on **irregular geometries**, but FE works ON any valid mesh
- **Neumann boundary conditions** are handled very naturally in FE—require an integral over surface, and nothing required for zero-Neumann BCs. Much more difficult in FD
- **Error analysis**

The finite difference method

Compare the finite difference equation:

$$\mathbf{U}^{n+1} = \mathbf{U}^n + \frac{\Delta t}{h^2} D\mathbf{U}^n + \Delta t\mathbf{F}^n$$

with the equivalent finite element equation:

$$M\mathbf{U}^{n+1} = M\mathbf{U}^n + \Delta t\, K\mathbf{U}^n + \Delta t\mathbf{b}^n$$

In fact, for a regular grid in 1D and with linear basis functions, $K = D/h$ (except for first/last rows).

The big advantages of FE over FD are

- FD is difficult to write down on **irregular geometries**, but FE works ON any valid mesh
- **Neumann boundary conditions** are handled very naturally in FE—require an integral over surface, and nothing required for zero-Neumann BCs. Much more difficult in FD
- Error analysis

The finite difference method

Compare the finite difference equation:

$$\mathbf{U}^{n+1} = \mathbf{U}^{n} + \frac{\Delta t}{h^2} D\mathbf{U}^{n} + \Delta t\mathbf{F}^{n}$$

with the equivalent finite element equation:

$$M\mathbf{U}^{n+1} = M\mathbf{U}^{n} + \Delta t\, K\mathbf{U}^{n} + \Delta t\mathbf{b}^{n}$$

In fact, for a regular grid in 1D and with linear basis functions, $K = D/h$ (except for first/last rows).

The big advantages of FE over FD are

- FD is difficult to write down on **irregular geometries**, but FE works ON any valid mesh
- **Neumann boundary conditions** are handled very naturally in FE— require an integral over surface, and nothing required for zero-Neumann BCs. Much more difficult in FD
- **Error analysis**

Finite volume methods

- Very commonly used for hyperbolic PDEs (for which the FE method tends to have trouble) and in computational fluid dynamics
- As with FE, FV is based on *integral formulations*.
- The domain is broken down into *control volumes* (similar to 'elements').
- One unknown computed per element (i.e. no need for 'node')—this can be considered to be the average value of u in the control volume.

Consider the advection equation $u_t + \nabla \cdot \mathbf{f}(u) = 0$. Integrate over a control volume Ω_i of volume V_i:

$$\int_{\Omega_i} u_t \,\mathrm{d}V = \int_{\Omega_i} -\nabla \cdot \mathbf{f}(u)\,\mathrm{d}V = -\int_{\partial\Omega_i} \mathbf{f}(u)\cdot\mathbf{n}\,\mathrm{d}S$$

Using an explicit time-discretisation, and $\int_{\Omega_i} U^n\,\mathrm{d}V \approx V_i U_i^n$, we obtain

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{V_i}\int_{\partial\Omega_i} \mathbf{f}(u^n)\cdot\mathbf{n}\,\mathrm{d}S$$

See eg `http://www.comp.leeds.ac.uk/meh/Talks/FVTutorial.pdf` for more details

Finite volume methods

- Very commonly used for hyperbolic PDEs (for which the FE method tends to have trouble) and in computational fluid dynamics
- As with FE, FV is based on *integral formulations*.
- The domain is broken down into *control volumes* (similar to 'elements').
- One unknown computed per element (i.e. no need for 'node')—this can be considered to be the average value of u in the control volume.

Consider the advection equation $u_t + \nabla \cdot \mathbf{f}(u) = 0$. Integrate over a control volume Ω_i of volume V_i:

$$\int_{\Omega_i} u_t \, \mathrm{d}V = \int_{\Omega_i} -\nabla \cdot \mathbf{f}(u) \, \mathrm{d}V = -\int_{\partial\Omega_i} \mathbf{f}(u) \cdot \mathbf{n} \, \mathrm{d}S$$

Using an explicit time-discretisation, and $\int_{\Omega_i} U^n \, \mathrm{d}V \approx V_i U_i^n$, we obtain

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{V_i} \int_{\partial\Omega_i} \mathbf{f}(u^n) \cdot \mathbf{n} \, \mathrm{d}S$$

See eg `http://www.comp.leeds.ac.uk/meh/Talks/FVTutorial.pdf` for more details

Finite volume methods

- Very commonly used for hyperbolic PDEs (for which the FE method tends to have trouble) and in computational fluid dynamics
- As with FE, FV is based on *integral formulations*.
- The domain is broken down into *control volumes* (similar to 'elements').
- One unknown computed per element (i.e. no need for 'node')—this can be considered to be the average value of u in the control volume.

Consider the advection equation $u_t + \nabla \cdot \mathbf{f}(u) = 0$. Integrate over a control volume Ω_i of volume V_i:

$$\int_{\Omega_i} u_t \, \mathrm{d}V = \int_{\Omega_i} -\nabla \cdot \mathbf{f}(u) \, \mathrm{d}V = -\int_{\partial\Omega_i} \mathbf{f}(u) \cdot \mathbf{n} \, \mathrm{d}S$$

Using an explicit time-discretisation, and $\int_{\Omega_i} U^n \, \mathrm{d}V \approx V_i U_i^n$, we obtain

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{V_i} \int_{\partial\Omega_i} \mathbf{f}(u^n) \cdot \mathbf{n} \, \mathrm{d}S$$

See eg `http://www.comp.leeds.ac.uk/meh/Talks/FVTutorial.pdf` for more details

Finite volume methods

- Very commonly used for hyperbolic PDEs (for which the FE method tends to have trouble) and in computational fluid dynamics
- As with FE, FV is based on *integral formulations*.
- The domain is broken down into *control volumes* (similar to 'elements').
- One unknown computed per element (i.e. no need for 'node')—this can be considered to be the average value of u in the control volume.

Consider the advection equation $u_t + \nabla \cdot \mathbf{f}(u) = 0$. Integrate over a control volume Ω_i of volume V_i:

$$\int_{\Omega_i} u_t \, \mathrm{d}V = \int_{\Omega_i} -\nabla \cdot \mathbf{f}(u) \, \mathrm{d}V = -\int_{\partial\Omega_i} \mathbf{f}(u) \cdot \mathbf{n} \, \mathrm{d}S$$

Using an explicit time-discretisation, and $\int_{\Omega_i} U^n \, \mathrm{d}V \approx V_i U_i^n$, we obtain

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{V_i} \int_{\partial\Omega_i} \mathbf{f}(u^n) \cdot \mathbf{n} \, \mathrm{d}S$$

See eg `http://www.comp.leeds.ac.uk/meh/Talks/FVTutorial.pdf` for more details

Finite volume methods

- Very commonly used for hyperbolic PDEs (for which the FE method tends to have trouble) and in computational fluid dynamics
- As with FE, FV is based on *integral formulations*.
- The domain is broken down into *control volumes* (similar to 'elements').
- One unknown computed per element (i.e. no need for 'node')—this can be considered to be the average value of u in the control volume.

Consider the advection equation $u_t + \nabla \cdot \mathbf{f}(u) = 0$. Integrate over a control volume Ω_i of volume V_i:

$$\int_{\Omega_i} u_t \,\mathrm{d}V = \int_{\Omega_i} -\nabla \cdot \mathbf{f}(u)\,\mathrm{d}V = -\int_{\partial\Omega_i} \mathbf{f}(u)\cdot\mathbf{n}\,\mathrm{d}S$$

Using an explicit time-discretisation, and $\int_{\Omega_i} U^n \,\mathrm{d}V \approx V_i U_i^n$, we obtain

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{V_i}\int_{\partial\Omega_i} \mathbf{f}(u^n)\cdot\mathbf{n}\,\mathrm{d}S$$

See eg `http://www.comp.leeds.ac.uk/meh/Talks/FVTutorial.pdf` for more details

Finite volume methods

- Very commonly used for hyperbolic PDEs (for which the FE method tends to have trouble) and in computational fluid dynamics
- As with FE, FV is based on *integral formulations*.
- The domain is broken down into *control volumes* (similar to 'elements').
- One unknown computed per element (i.e. no need for 'node')—this can be considered to be the average value of u in the control volume.

Consider the advection equation $u_t + \nabla \cdot \mathbf{f}(u) = 0$. Integrate over a control volume Ω_i of volume V_i:

$$\int_{\Omega_i} u_t \, \mathrm{d}V = \int_{\Omega_i} -\nabla \cdot \mathbf{f}(u) \, \mathrm{d}V = -\int_{\partial\Omega_i} \mathbf{f}(u) \cdot \mathbf{n} \, \mathrm{d}S$$

Using an explicit time-discretisation, and $\int_{\Omega_i} U^n \, \mathrm{d}V \approx V_i U_i^n$, we obtain

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{V_i} \int_{\partial\Omega_i} \mathbf{f}(u^n) \cdot \mathbf{n} \, \mathrm{d}S$$

See eg `http://www.comp.leeds.ac.uk/meh/Talks/FVTutorial.pdf` for more details

Methods of weight residuals

In FE we used an *integral formulation of the PDE*, eg: *find* $u \in \mathcal{V}$ *such that:*

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}V = \int_\Omega fv \,\mathrm{d}V + \int_{\Gamma_2} gv \,\mathrm{d}S \quad \forall v \in \mathcal{V}$$

Write this as: *find* $u \in \mathcal{V}$ *such that:* $a(u, v) = l(v) \quad \forall v \in \mathcal{V}$

To *discretise* the integral equation, we replace $\mathcal{V}$ by *finite-dimensional subspaces* (of dimension N): *find* $u^{\text{approx}} \in \mathcal{W}_1$ *such that:*

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

Choosing bases:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \ldots, \phi_N\} \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \ldots, \chi_N\} \end{aligned}$$

(ie $u^{\text{approx}} = \sum \alpha_i \phi_i$), we can obtain N equations for N unknowns.

Different methods are based on different choices of $\mathcal{W}_1$ and $\mathcal{W}_2$.

Methods of weight residuals

In FE we used an *integral formulation of the PDE*, eg: *find* $u \in \mathcal{V}$ *such that:*

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}V = \int_\Omega fv \, \mathrm{d}V + \int_{\Gamma_2} gv \, \mathrm{d}S \quad \forall v \in \mathcal{V}$$

Write this as: *find* $u \in \mathcal{V}$ *such that:* $a(u, v) = l(v) \quad \forall v \in \mathcal{V}$

To *discretise* the integral equation, we replace $\mathcal{V}$ by *finite-dimensional subspaces* (of dimension N): *find* $u^{approx} \in \mathcal{W}_1$ *such that:*

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

Choosing bases:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \dots, \phi_N\} \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \dots, \chi_N\} \end{aligned}$$

(ie $u^{\text{approx}} = \sum \alpha_i \phi_i$), we can obtain N equations for N unknowns.

Different methods are based on different choices of $\mathcal{W}_1$ and $\mathcal{W}_2$.

Methods of weight residuals

In FE we used an *integral formulation of the PDE*, eg: *find $u \in \mathcal{V}$ such that:*

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}V = \int_\Omega fv \,\mathrm{d}V + \int_{\Gamma_2} gv \,\mathrm{d}S \quad \forall v \in \mathcal{V}$$

Write this as: *find $u \in \mathcal{V}$ such that:* $a(u, v) = l(v) \quad \forall v \in \mathcal{V}$

To *discretise* the integral equation, we replace $\mathcal{V}$ by *finite-dimensional subspaces* (of dimension N): *find $u^{approx} \in \mathcal{W}_1$ such that:*

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

Choosing bases:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \ldots, \phi_N\} \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \ldots, \chi_N\} \end{aligned}$$

(ie $u^{\text{approx}} = \sum \alpha_i \phi_i$), we can obtain N equations for N unknowns.

Different methods are based on different choices of $\mathcal{W}_1$ and $\mathcal{W}_2$.

Methods of weight residuals

In FE we used an *integral formulation of the PDE*, eg: *find* $u \in \mathcal{V}$ *such that:*

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}V = \int_\Omega fv \, \mathrm{d}V + \int_{\Gamma_2} gv \, \mathrm{d}S \quad \forall v \in \mathcal{V}$$

Write this as: *find* $u \in \mathcal{V}$ *such that:* $a(u, v) = l(v) \quad \forall v \in \mathcal{V}$

To *discretise* the integral equation, we replace $\mathcal{V}$ by *finite-dimensional subspaces* (of dimension N): *find* $u^{approx} \in \mathcal{W}_1$ *such that:*

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

Choosing bases:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \ldots, \phi_N\} \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \ldots, \chi_N\} \end{aligned}$$

(ie $u^{\text{approx}} = \sum \alpha_i \phi_i$), we can obtain N equations for N unknowns.

Different methods are based on different choices of $\mathcal{W}_1$ and $\mathcal{W}_2$.

Methods of weight residuals

Find $u^{\text{approx}} \in \mathcal{W}_1$ such that:

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

with:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \ldots, \phi_N\}. \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \ldots, \chi_N\} \end{aligned}$$

- **Galerkin methods**: use $\phi = \chi$, i.e. $\mathcal{W}_1 = \mathcal{W}_2$
- **Collocation methods**: use δ-functions for χ's (i.e. replace integrals with point evaluations (at N collocation points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$)
- **(Continuous) Galerkin FEM**: use $\mathcal{W}_1 = \mathcal{W}_2$ and take the ϕ_i to be continuous and **piecewise polynomial** on element
  - As we know in practice we just consider 1 canonical element and define the basis functions on this (the *shape functions*)
  - Elements could be tetrahedral/hexahedral, shape functions could be linear, quadratic, cubic Hermite and more...
- **Discontinuous Galerkin FEM**: ϕ_i piecewise polynomial but no longer continuous across elements
- **Spectral methods**: ϕ_k globally continuous and infinitely differentiable (for example, $\phi_k(x) = exp(ikx)$)

Methods of weight residuals

Find $u^{\text{approx}} \in \mathcal{W}_1$ such that:

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

with:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \ldots, \phi_N\}. \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \ldots, \chi_N\} \end{aligned}$$

- **Galerkin methods**: use $\phi = \chi$, i.e. $\mathcal{W}_1 = \mathcal{W}_2$
- **Collocation methods**: use δ-functions for χ's (i.e. replace integrals with point evaluations (at N collocation points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$)
- **(Continuous) Galerkin FEM**: use $\mathcal{W}_1 = \mathcal{W}_2$ and take the ϕ_i to be continuous and **piecewise polynomial** on element
    - As we know in practice we just consider 1 canonical element and define the basis functions on this (the *shape functions*)
    - Elements could be tetrahedral/hexahedral, shape functions could be linear, quadratic, cubic Hermite and more...
- **Discontinuous Galerkin FEM**: ϕ_i piecewise polynomial but no longer continuous across elements
- **Spectral methods**: ϕ_k globally continuous and infinitely differentiable (for example, $\phi_k(x) = exp(ikx)$)

Methods of weight residuals

Find $u^{\text{approx}} \in \mathcal{W}_1$ such that:

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

with:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \dots, \phi_N\}. \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \dots, \chi_N\} \end{aligned}$$

- **Galerkin methods**: use $\phi = \chi$, i.e. $\mathcal{W}_1 = \mathcal{W}_2$
- **Collocation methods**: use δ-functions for χ's (i.e. replace integrals with point evaluations (at N collocation points $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$)
- **(Continuous) Galerkin FEM**: use $\mathcal{W}_1 = \mathcal{W}_2$ and take the ϕ_i to be continuous and **piecewise polynomial** on element
  - As we know in practice we just consider 1 canonical element and define the basis functions on this (the *shape functions*)
  - Elements could be tetrahedral/hexahedral, shape functions could be linear, quadratic, cubic Hermite and more...
- **Discontinuous Galerkin FEM**: ϕ_i piecewise polynomial but no longer continuous across elements
- **Spectral methods**: ϕ_k globally continuous and infinitely differentiable (for example, $\phi_k(x) = exp(ikx)$)

Methods of weight residuals

Find $u^{\text{approx}} \in \mathcal{W}_1$ such that:

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

with:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \ldots, \phi_N\}. \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \ldots, \chi_N\} \end{aligned}$$

- **Galerkin methods**: use $\phi = \chi$, i.e. $\mathcal{W}_1 = \mathcal{W}_2$
- **Collocation methods**: use δ-functions for χ's (i.e. replace integrals with point evaluations (at N collocation points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$)
- **(Continuous) Galerkin FEM**: use $\mathcal{W}_1 = \mathcal{W}_2$ and take the ϕ_i to be continuous and **piecewise polynomial** on element
    - As we know in practice we just consider 1 canonical element and define the basis functions on this (the *shape functions*)
    - Elements could be tetrahedral/hexahedral, shape functions could be linear, quadratic, cubic Hermite and more...
- **Discontinuous Galerkin FEM**: ϕ_i piecewise polynomial but no longer continuous across elements
- **Spectral methods**: ϕ_k globally continuous and infinitely differentiable (for example, $\phi_k(x) = exp(ikx)$)

Methods of weight residuals

Find $u^{\text{approx}} \in \mathcal{W}_1$ such that:

$$a(u^{\text{approx}}, v) = l(v) \quad \forall v \in \mathcal{W}_2$$

with:

$$\begin{aligned} \mathcal{W}_1 &= \text{span}\{\phi_1, \dots, \phi_N\}. \\ \mathcal{W}_2 &= \text{span}\{\chi_1, \dots, \chi_N\} \end{aligned}$$

- **Galerkin methods**: use $\phi = \chi$, i.e. $\mathcal{W}_1 = \mathcal{W}_2$
- **Collocation methods**: use δ-functions for χ's (i.e. replace integrals with point evaluations (at N collocation points $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$)
- **(Continuous) Galerkin FEM**: use $\mathcal{W}_1 = \mathcal{W}_2$ and take the ϕ_i to be continuous and **piecewise polynomial** on element
  - As we know in practice we just consider 1 canonical element and define the basis functions on this (the *shape functions*)
  - Elements could be tetrahedral/hexahedral, shape functions could be linear, quadratic, cubic Hermite and more...
- **Discontinuous Galerkin FEM**: ϕ_i piecewise polynomial but no longer continuous across elements
- **Spectral methods**: ϕ_k globally continuous and infinitely differentiable (for example, $\phi_k(x) = exp(ikx)$)

Spectral methods

- There are both spectral-collocation methods (work with the strong form) or spectral-Galerkin methods (work with the weak form)

- Various choices of basis functions ($\mathcal{W}_1$) are possible, for example
- For problems with **periodic boundary conditions**, use $\phi_k(x) = exp(ikx)$
  - i.e. $u^{\text{approx}} = \sum \alpha_k \phi_k$ approximates u with a cut-off Fourier series
- For problems with **non-periodic boundary conditions**: use a set of 'orthogonal polynomials' for ϕ_k, such a Legendre or Chebychev polynomials

- For problems with *smooth data* (initial condition, boundary conditions, forces etc are smooth functions), spectral methods give exceptional rates of convergence.
- For more info, see e.g. `http://www.lorene.obspm.fr/palma.pdf`

Spectral methods

- There are both spectral-collocation methods (work with the strong form) or spectral-Galerkin methods (work with the weak form)

- Various choices of basis functions ($\mathcal{W}_1$) are possible, for example
- For problems with **periodic boundary conditions**, use $\phi_k(x) = exp(ikx)$
  - i.e. $u^{\text{approx}} = \sum \alpha_k \phi_k$ approximates u with a cut-off Fourier series
- For problems with **non-periodic boundary conditions**: use a set of 'orthogonal polynomials' for ϕ_k, such a Legendre or Chebychev polynomials

- For problems with *smooth data* (initial condition, boundary conditions, forces etc are smooth functions), spectral methods give exceptional rates of convergence.

- For more info, see e.g. `http://www.lorene.obspm.fr/palma.pdf`

Spectral methods

- There are both spectral-collocation methods (work with the strong form) or spectral-Galerkin methods (work with the weak form)

- Various choices of basis functions ($\mathcal{W}_1$) are possible, for example
- For problems with **periodic boundary conditions**, use $\phi_k(x) = exp(ikx)$
  - i.e. $u^{\text{approx}} = \sum \alpha_k \phi_k$ approximates u with a cut-off Fourier series
- For problems with **non-periodic boundary conditions**: use a set of 'orthogonal polynomials' for ϕ_k, such a Legendre or Chebychev polynomials

- For problems with *smooth data* (initial condition, boundary conditions, forces etc are smooth functions), spectral methods give exceptional rates of convergence.
- For more info, see e.g. `http://www.lorene.obspm.fr/palma.pdf`

Continuum Mechanics

Overview

* Focussing on nonlinear elasticity, but also mentioning linear elasticity & fluids
** Nonlinear elasticity only

Overview

* Focussing on nonlinear elasticity, but also mentioning linear elasticity & fluids
** Nonlinear elasticity only

Overview

1. Introduction: solids and fluids
2. Kinematics*
3. Balance equations*
4. Material laws*
5. Overall governing equations*
6. Weak problem and numerical method**
7. Objected-oriented design in Chaste**

* Focussing on nonlinear elasticity, but also mentioning linear elasticity & fluids
** Nonlinear elasticity only

Overview

* Focussing on nonlinear elasticity, but also mentioning linear elasticity & fluids
** Nonlinear elasticity only

Overview

1. Introduction: solids and fluids
2. Kinematics*
3. Balance equations*
4. Material laws*
5. Overall governing equations*
6. Weak problem and numerical method**
7. Objected-oriented design in Chaste**

* Focussing on nonlinear elasticity, but also mentioning linear elasticity & fluids
** Nonlinear elasticity only

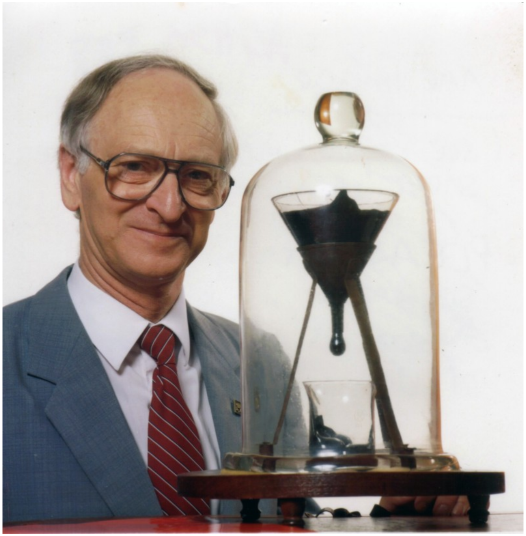
Overview

* Focussing on nonlinear elasticity, but also mentioning linear elasticity & fluids
** Nonlinear elasticity only

Introduction: solids and fluids

Solids versus fluids

Solids versus fluids



Pitchdrop experiment, Queensland. Experiment begun 1927 (1930). Drops fell in: 1938, 1947, 1954, 1962, 1970, 1979, 1988, 2000

Solids versus fluids

“Fluids cannot resist deformation force”. Shape will change as long as the force is applied. Whereas a solid can change shape but not indefinitely.

More specifically, fluids cannot resist shear forces

- For solids, force is related to *deformation* (coefficient: *stiffness*)
- For fluids, force is related to *deformation-rate* (coefficient: *viscosity*)

Some materials are fluid under some conditions (excl. temperature) and solid under others (see, for example, `youtube:walking on custard`)

Solids versus fluids

"Fluids cannot resist deformation force". Shape will change as long as the force is applied. Whereas a solid can change shape but not indefinitely.

More specifically, fluids cannot resist shear forces

- For solids, force is related to *deformation* (coefficient: *stiffness*)
- For fluids, force is related to *deformation-rate* (coefficient: *viscosity*)

Some materials are fluid under some conditions (excl. temperature) and solid under others (see, for example, `youtube:walking on custard`)

Types of solid

For solids, force is related to deformation (stress related to strain):

Elastic

- When an applied force is removed, the solid returns to its original shape
- For small enough forces/strains, stress is usually proportional to strain (*linear elasticity*)

Visco-elastic

- Also exhibit a *viscous response*, for example, slow change of shape if a force is held constant / slow decrease of stress if strain held constant
- Stress becomes a function of strain and *strain rate*

Plastic

- Once a large enough stress is applied (the *yield* stress), the material undergoes permanent deformation (*flows*), due to internal rearrangement. If the stress is removed it won't return back to original state.

Types of solid

For solids, force is related to deformation (stress related to strain):

Elastic

- When an applied force is removed, the solid returns to its original shape
- For small enough forces/strains, stress is usually proportional to strain (*linear elasticity*)

Visco-elastic

- Also exhibit a *viscous response*, for example, slow change of shape if a force is held constant / slow decrease of stress if strain held constant
- Stress becomes a function of strain and *strain rate*

Plastic

- Once a large enough stress is applied (the *yield* stress), the material undergoes permanent deformation (*flows*), due to internal rearrangement. If the stress is removed it won't return back to original state.

Types of solid

For solids, force is related to deformation (stress related to strain):

Elastic

- When an applied force is removed, the solid returns to its original shape
- For small enough forces/strains, stress is usually proportional to strain (*linear elasticity*)

Visco-elastic

- Also exhibit a *viscous response*, for example, slow change of shape if a force is held constant / slow decrease of stress if strain held constant
- Stress becomes a function of strain and *strain rate*

Plastic

- Once a large enough stress is applied (the *yield* stress), the material undergoes permanent deformation (*flows*), due to internal rearrangement. If the stress is removed it won't return back to original state.

Types of fluid

For fluids force is related to deformation-rate (stress is related to strain-rate)

Newtonian

- Stress is related *linearly* to the strain-rate

Non-Newtonian

- Stress is related *non-linearly* to the strain-rate

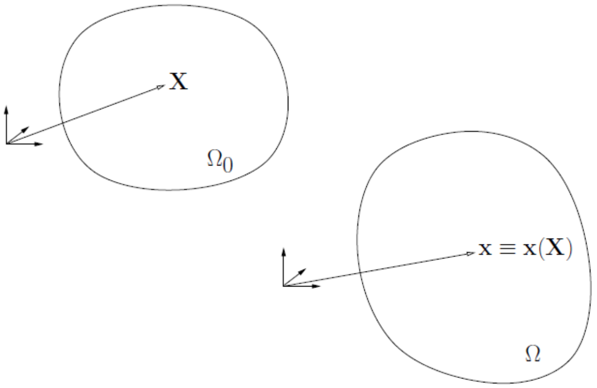
Kinematics of solids

Kinematics of solids

In the later section we will write down balance equations relating the internal stresses in the body to external forces.

What are the internal stresses a function of?

Undeformed and deformed states



- Let Ω_0 represent the unloaded, unstressed body
- Let Ω_t represent the deformed body at time t
  - For time-independent problems, we denote the deformed body Ω
- Let $\mathbf{X}$ represent a point in the undeformed body
- Let $\mathbf{x} \equiv \mathbf{x}(t, \mathbf{X})$ represent the corresponding deformed position
- Let the displacement be denoted $\mathbf{u} = \mathbf{x} - \mathbf{X}$

The deformation gradient

Let $F_{iM} = \frac{\partial x_i}{\partial X_M}$ be the deformation gradient. This describes the deformation, excluding rigid body translations.

Any deformation can be decomposed into a (local) translation, rotation, and stretch. Correspondingly, F can be decomposed into a rotation and a stretch: $F = RU$, where R is an rotation matrix, and U is a positive-definite symmetric matrix representing stretch.

Examples, in 2D:

- let $\mathbf{x} = \begin{bmatrix} \alpha X \\ \beta Y \end{bmatrix}$, then $F = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$ (simple bi-axial stretch)
- let $\mathbf{x} = \begin{bmatrix} X - \alpha Y \\ Y \end{bmatrix}$, then $F = \begin{bmatrix} 1 & -\alpha \\ 0 & 1 \end{bmatrix}$ (simple shear)

The deformation gradient

Let $F_{iM} = \frac{\partial x_i}{\partial X_M}$ be the deformation gradient. This describes the deformation, excluding rigid body translations.

Any deformation can be decomposed into a (local) translation, rotation, and stretch. Correspondingly, F can be decomposed into a rotation and a stretch: $F = RU$, where R is an rotation matrix, and U is a positive-definite symmetric matrix representing stretch.

Examples, in 2D:

- let $\mathbf{x} = \begin{bmatrix} \alpha X \\ \beta Y \end{bmatrix}$, then $F = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$ (simple bi-axial stretch)
- let $\mathbf{x} = \begin{bmatrix} X - \alpha Y \\ Y \end{bmatrix}$, then $F = \begin{bmatrix} 1 & -\alpha \\ 0 & 1 \end{bmatrix}$ (simple shear)

The deformation gradient

Let $F_{iM} = \frac{\partial x_i}{\partial X_M}$ be the deformation gradient. This describes the deformation, excluding rigid body translations.

Any deformation can be decomposed into a (local) translation, rotation, and stretch. Correspondingly, F can be decomposed into a rotation and a stretch: $F = RU$, where R is an rotation matrix, and U is a positive-definite symmetric matrix representing stretch.

Examples, in 2D:

- let $\mathbf{x} = \begin{bmatrix} \alpha X \\ \beta Y \end{bmatrix}$, then $F = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$ (simple bi-axial stretch)
- let $\mathbf{x} = \begin{bmatrix} X - \alpha Y \\ Y \end{bmatrix}$, then $F = \begin{bmatrix} 1 & -\alpha \\ 0 & 1 \end{bmatrix}$ (simple shear)

The deformation gradient

det F

F is the jacobian of the mapping from Ω_0 to Ω, therefore det F represents the change in local volume. Hence:

- det $F > 0$ for all deformations
- For incompressible deformations (also known as *isochoric* or *isovolumetric* deformations), det $F = 1$ (everywhere)

Define $J = \det F$

Principal stretches

The eigenvalues of U are of the *principal stretches*, denoted $\lambda_1, \lambda_2, \lambda_3$

The deformation gradient

$\det F$

F is the jacobian of the mapping from Ω_0 to Ω, therefore $\det F$ represents the change in local volume. Hence:

- $\det F > 0$ for all deformations
- For incompressible deformations (also known as *isochoric* or *isovolumetric* deformations), $\det F = 1$ (everywhere)

Define $J = \det F$

Principal stretches

The eigenvalues of U are of the *principal stretches*, denoted $\lambda_1, \lambda_2, \lambda_3$

Lagrangian measures of strain

The *(right) Cauchy-Green deformation tensor* is

$$C = F^{\mathsf{T}} F$$

Note that

- $F = RU \quad \Rightarrow \quad C = U^{\mathsf{T}} R^{\mathsf{T}} RU = U^{\mathsf{T}} U = U^2$
- i.e. C is independent of the rotation
- the eigenvalues of C are λ_1^2, λ_2^2, λ_3^2.

The *Green-Lagrange strain tensor* is

$$E = \frac{1}{2}(C - I)$$

and is the nonlinear generalisation of the simple 1d strain measure $(l - l_0)/l_0$

Can work with either C or E. Note: for no deformation $C = I$ vs $E = 0$.

Lagrangian measures of strain

The *(right) Cauchy-Green deformation tensor* is

$$C = F^{\mathsf{T}} F$$

Note that

- $F = RU \quad \Rightarrow \quad C = U^{\mathsf{T}} R^{\mathsf{T}} R U = U^{\mathsf{T}} U = U^2$
- i.e. C is independent of the rotation
- the eigenvalues of C are $\lambda_1^2,\ \lambda_2^2,\ \lambda_3^2$.

The *Green-Lagrange strain tensor* is

$$E = \frac{1}{2}(C - I)$$

and is the nonlinear generalisation of the simple 1d strain measure $(l - l_0)/l_0$

Can work with either C or E. Note: for no deformation $C = I$ vs $E = 0$.

Lagrangian measures of strain

The *(right) Cauchy-Green deformation tensor* is

$$C = F^{\mathsf{T}} F$$

Note that

- $F = RU \quad \Rightarrow \quad C = U^{\mathsf{T}} R^{\mathsf{T}} R U = U^{\mathsf{T}} U = U^2$
- i.e. C is independent of the rotation
- the eigenvalues of C are $\lambda_1^2,\ \lambda_2^2,\ \lambda_3^2$.

The *Green-Lagrange strain tensor* is

$$E = \frac{1}{2}(C - I)$$

and is the nonlinear generalisation of the simple 1d strain measure $(l - l_0)/l_0$

Can work with either C or E. Note: for no deformation $C = I$ vs $E = 0$.

Lagrangian measures of strain

The *(right) Cauchy-Green deformation tensor* is

$$C = F^{\mathsf{T}} F$$

Note that

- $F = RU \quad \Rightarrow \quad C = U^{\mathsf{T}} R^{\mathsf{T}} R U = U^{\mathsf{T}} U = U^2$
- i.e. C is independent of the rotation
- the eigenvalues of C are $\lambda_1^2,\ \lambda_2^2,\ \lambda_3^2$.

The *Green-Lagrange strain tensor* is

$$E = \frac{1}{2}(C - I)$$

and is the nonlinear generalisation of the simple 1d strain measure $(l - l_0)/l_0$

Can work with either C or E. Note: for no deformation $C = I$ vs $E = 0$.

Isotropic problems

We shall see later how stress is a function of strain E, or equivalently, of C, say $\sigma = \sigma(C_{11}, C_{12}, C_{13}, C_{22}, C_{23}, C_{33})$.

If can be shown that for *isotropic* problems, the stress is just a function of the *principal invariants*[4] of C

$$\begin{aligned} I_1 &= \mathrm{tr}(C) \\ I_2 &= \frac{1}{2}\left(\mathrm{tr}(C)^2 - \mathrm{tr}(C^2)\right) \\ I_3 &= \det(C) \end{aligned}$$

[4] To complicate matters even more, compressible problems often use the *deviatoric invariants*: $\bar{I}_1 = I_1 I_3^{-\frac{1}{3}}$, and $\bar{I}_2 = I_2 I_3^{-\frac{2}{3}}$. These are the invariants of C after it has been scaled to have determinant 1—see [Horgan and Saccomandi, Journal of Elasticity, 2004] for a discussion.

Isotropic problems

We shall see later how stress is a function of strain E, or equivalently, of C, say $\sigma = \sigma(C_{11}, C_{12}, C_{13}, C_{22}, C_{23}, C_{33})$.

If can be shown that for *isotropic* problems, the stress is just a function of the *principal invariants*[4] of C

$$\begin{aligned} I_1 &= \mathrm{tr}(C) \\ I_2 &= \frac{1}{2}\left(\mathrm{tr}(C)^2 - \mathrm{tr}(C^2)\right) \\ I_3 &= \det(C) \end{aligned}$$

[4]To complicate matters even more, compressible problems often use the *deviatoric invariants*: $\bar{I}_1 = I_1 I_3^{-\frac{1}{3}}$, and $\bar{I}_2 = I_2 I_3^{-\frac{2}{3}}$. These are the invariants of C after it has been scaled to have determinant 1—see [Horgan and Saccomandi, Journal of Elasticity, 2004] for a discussion.

Linear elasticity

Linearise E, removing terms that are quadratic in the displacement:

$$E_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i}\right) + \mathcal{O}(u^2)$$

so define

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i}\right)$$

This is the *infinitesimal strain tensor*

Note: normally in linear elasticity $\mathbf{x}$ represents undeformed position, so ϵ_{ij} is defined to be $\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$.

Note also that linearising the incompressibility constraint $\det F = 1$ gives:

$$\nabla \cdot u = 0$$

Linear elasticity

Linearise E, removing terms that are quadratic in the displacement:

$$E_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i}\right) + \mathcal{O}(u^2)$$

so define

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i}\right)$$

This is the *infinitesimal strain tensor*

Note: normally in linear elasticity **x** represents undeformed position, so ϵ_{ij} is defined to be $\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$.

Note also that linearising the incompressibility constraint $\det F = 1$ gives:

$$\nabla \cdot u = 0$$

Linear elasticity

Linearise E, removing terms that are quadratic in the displacement:

$$E_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i}\right) + \mathcal{O}(u^2)$$

so define

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial X_j} + \frac{\partial u_j}{\partial X_i}\right)$$

This is the *infinitesimal strain tensor*

Note: normally in linear elasticity **x** represents undeformed position, so ϵ_{ij} is defined to be $\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$.

Note also that linearising the incompressibility constraint $\det F = 1$ gives:

$$\nabla \cdot u = 0$$

Balance laws

Stresses

There are various definitions of stress in nonlinear elasticity:

Cauchy stress, σ, the force per unit deformed area acting on surfaces on the deformed body (i.e. the true stress) (symmetric)

1st Piola-Kirchhoff stress, S, the force per unit *undeformed area* acting on surfaces on the deformed body (not symmetric)

2nd Piola-Kirchhoff stress, T, the force per unit undeformed area acting on surfaces on the undeformed body (symmetric)

Relationships:

$$S = JF^{-1}\sigma \qquad T = SF^{-\mathrm{T}} \qquad \sigma = \frac{1}{J}FTF^{\mathrm{T}}$$

Stresses

There are various definitions of stress in nonlinear elasticity:

Cauchy stress, σ, the force per unit deformed area acting on surfaces on the deformed body (i.e. the true stress) (symmetric)

1st Piola-Kirchhoff stress, S, the force per unit *undeformed area* acting on surfaces on the deformed body (not symmetric)

2nd Piola-Kirchhoff stress, T, the force per unit undeformed area acting on surfaces on the undeformed body (symmetric)

Relationships:

$$S = JF^{-1}\sigma \qquad T = SF^{-\mathrm{T}} \qquad \sigma = \frac{1}{J}FTF^{\mathrm{T}}$$

Stresses

There are various definitions of stress in nonlinear elasticity:

Cauchy stress, σ, the force per unit deformed area acting on surfaces on the deformed body (i.e. the true stress) (symmetric)

1st Piola-Kirchhoff stress, S, the force per unit *undeformed area* acting on surfaces on the deformed body (not symmetric)

2nd Piola-Kirchhoff stress, T, the force per unit undeformed area acting on surfaces on the undeformed body (symmetric)

Relationships:

$$S = JF^{-1}\sigma \qquad T = SF^{-\mathrm{T}} \qquad \sigma = \frac{1}{J}FTF^{\mathrm{T}}$$

Stresses

There are various definitions of stress in nonlinear elasticity:

Cauchy stress, σ, the force per unit deformed area acting on surfaces on the deformed body (i.e. the true stress) (symmetric)

1st Piola-Kirchhoff stress, S, the force per unit *undeformed area* acting on surfaces on the deformed body (not symmetric)

2nd Piola-Kirchhoff stress, T, the force per unit undeformed area acting on surfaces on the undeformed body (symmetric)

Relationships:

$$S = JF^{-1}\sigma \qquad T = SF^{-\mathrm{T}} \qquad \sigma = \frac{1}{J}FTF^{\mathrm{T}}$$

Stresses

There are various definitions of stress in nonlinear elasticity:

Cauchy stress, σ, the force per unit deformed area acting on surfaces on the deformed body (i.e. the true stress) (symmetric)

1st Piola-Kirchhoff stress, S, the force per unit *undeformed area* acting on surfaces on the deformed body (not symmetric)

2nd Piola-Kirchhoff stress, T, the force per unit undeformed area acting on surfaces on the undeformed body (symmetric)

Relationships:

$$S = JF^{-1}\sigma \qquad T = SF^{-\mathrm{T}} \qquad \sigma = \frac{1}{J}FTF^{\mathrm{T}}$$

Equilibrium equation

Let ρ_0 and ρ be the density in the undeformed and deformed bodies, and let $\mathbf{b}$ be the *body force density* (e.g. gravity, for which $\mathbf{b} = [0, 0, -9.81]$)

For a body in *static equilibrium*, the equilibrium equation is

$$\frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i = 0 \qquad \text{in } \Omega$$

This isn't particularly useful as $\mathbf{x}$ and Ω are unknown

Equilibrium equation

Let ρ_0 and ρ be the density in the undeformed and deformed bodies, and let $\mathbf{b}$ be the *body force density* (e.g. gravity, for which $\mathbf{b} = [0, 0, -9.81]$)

For a body in *static equilibrium*, the equilibrium equation is

$$\frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i = 0 \qquad \text{in } \Omega$$

This isn't particularly useful as $\mathbf{x}$ and Ω are unknown

Equilibrium equation

Let ρ_0 and ρ be the density in the undeformed and deformed bodies, and let **b** be the *body force density* (e.g. gravity, for which $\mathbf{b} = [0, 0, -9.81]$)

For a body in *static equilibrium*, the equilibrium equation is

$$\frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i = 0 \qquad \text{in } \Omega$$

This isn't particularly useful as **x** and Ω are unknown

Equilibrium equation

We can transform to the undeformed state, for which the 1st Piola-Kirchhoff stress arises

$$\frac{\partial S_{Mi}}{\partial X_M} + \rho_0 b_i = 0 \qquad \text{in } \Omega_0$$

The two equations can be written as

$$\begin{aligned} \mathrm{div}(\sigma) + \rho \mathbf{b} &= 0 \qquad \text{in } \Omega \\ \mathrm{Div}(S) + \rho_0 \mathbf{b} &= 0 \qquad \text{in } \Omega_0 \end{aligned}$$

Replacing the 1st PK stress with the 2nd PK stress, we obtain

$$\frac{\partial}{\partial X_M}\left(T_{MN} \frac{\partial x_i}{\partial X_N} \right) + \rho_0 b_i = 0 \qquad \text{in } \Omega_0$$

Note that $T \equiv T(\mathbf{x})$ (through some material-dependent relationship, to be discussed)

Equilibrium equation

We can transform to the undeformed state, for which the 1st Piola-Kirchhoff stress arises

$$\frac{\partial S_{Mi}}{\partial X_M} + \rho_0 b_i = 0 \qquad \text{in } \Omega_0$$

The two equations can be written as

$$\begin{aligned} \mathrm{div}(\sigma) + \rho \mathbf{b} &= 0 \qquad \text{in } \Omega \\ \mathrm{Div}(S) + \rho_0 \mathbf{b} &= 0 \qquad \text{in } \Omega_0 \end{aligned}$$

Replacing the 1st PK stress with the 2nd PK stress, we obtain

$$\frac{\partial}{\partial X_M}\left(T_{MN} \frac{\partial x_i}{\partial X_N} \right) + \rho_0 b_i = 0 \qquad \text{in } \Omega_0$$

Note that $T \equiv T(\mathbf{x})$ (through some material-dependent relationship, to be discussed)

Boundary conditions

Mixed Dirichlet-Neumann boundary conditions are the specification of

- deformation/displacement on one part of the boundary (Dirichlet BCs)
- *tractions* the rest of the boundary (Neumann BCs)

The Neumann boundary condition is

$$\sigma_{ij} n_j = s_i \qquad \text{on deformed surface}$$

where $\mathbf{s}$ is the prescribed traction, which again has to be transformed back to the undeformed body

Splitting $\partial\Omega_0$ into Γ_1 and Γ_2, overall the boundary conditions are

$$\begin{aligned} \mathbf{x} &= \mathbf{x}^* \qquad && \text{on } \Gamma_1 \\ S_{Mi} N_M &= s_i \qquad && \text{on } \Gamma_2 \end{aligned}$$

(and again we could replace S with TF^{T})

Boundary conditions

Mixed Dirichlet-Neumann boundary conditions are the specification of

- deformation/displacement on one part of the boundary (Dirichlet BCs)
- *tractions* the rest of the boundary (Neumann BCs)

The Neumann boundary condition is

$$\sigma_{ij} n_j = s_i \qquad \text{on deformed surface}$$

where **s** is the prescribed traction, which again has to be transformed back to the undeformed body

Splitting $\partial\Omega_0$ into Γ_1 and Γ_2, overall the boundary conditions are

$$\mathbf{x} = \mathbf{x}^* \qquad \text{on } \Gamma_1$$
$$S_{Mi} N_M = s_i \qquad \text{on } \Gamma_2$$

(and again we could replace S with TF^{T})

Boundary conditions

Mixed Dirichlet-Neumann boundary conditions are the specification of

- deformation/displacement on one part of the boundary (Dirichlet BCs)
- *tractions* the rest of the boundary (Neumann BCs)

The Neumann boundary condition is

$$\sigma_{ij} n_j = s_i \qquad \text{on deformed surface}$$

where **s** is the prescribed traction, which again has to be transformed back to the undeformed body

Splitting $\partial\Omega_0$ into Γ_1 and Γ_2, overall the boundary conditions are

$$\begin{aligned} \mathbf{x} &= \mathbf{x}^* && \text{on } \Gamma_1 \\ S_{Mi} N_M &= s_i && \text{on } \Gamma_2 \end{aligned}$$

(and again we could replace S with TF^{T})

Other balance equations

Linear elasticity

Use $\frac{\partial}{\partial x_i} \approx \frac{\partial}{\partial X_i}$ (which means all 3 types of stress are equal to lowest order); work with

$$\frac{\partial \sigma_{ij}}{\partial X_j} + \rho_0 b_i = 0 \qquad \text{in } \Omega_0$$

Time-dependent problems

Defining the acceleration $\mathbf{a} = \frac{\partial^2 \mathbf{x}}{\partial t^2}$.

$$\rho_0 a_i = \frac{\partial}{\partial X_M}\left(T_{MN} \frac{\partial x_i}{\partial X_N} \right) + \rho_0 b_i \qquad \text{in } \Omega_0$$

Material laws

The strain energy function

An *elastic* material is one where stress is a function of strain: $T \equiv T(E)$ say

A **hyper-elastic** material is an elastic material for which there exists a **strain energy function** whose *derivative with respect to strain gives the stress*.

Specifically, there exists $W \equiv W(E)$ such that[5]

$$T_{MN} = \frac{\partial W}{\partial E_{MN}}$$

W must be determined experimentally (propose a law and experimentally determine parameters)

[5]For reference, it is also the case that $S_{Mi} = \frac{\partial W}{\partial F_{iM}}$

The strain energy function

An *elastic* material is one where stress is a function of strain: $T \equiv T(E)$ say

A **hyper-elastic** material is an elastic material for which there exists a **strain energy function** whose *derivative with respect to strain gives the stress*.

Specifically, there exists $W \equiv W(E)$ such that[5]

$$T_{MN} = \frac{\partial W}{\partial E_{MN}}$$

W must be determined experimentally (propose a law and experimentally determine parameters)

[5] For reference, it is also the case that $S_{Mi} = \frac{\partial W}{\partial F_{iM}}$

The strain energy function

It is often simpler to just work with C

$$W \equiv W(C) \qquad \text{such that} \qquad T_{MN} = 2\frac{\partial W}{\partial C_{MN}}$$

Isotropic materials

- In general, W is a function of the six independent components of C (recall that C is symmetric)
- However, for (compressible) *isotropic* materials, it can be shown that

$$W \equiv W(I_1, I_2, I_3)$$

only

Incompressible strain energy functions

Recall that for incompressible materials we have the constraint

$$\det F = 1$$

(everywhere), i.e. $I_3 = 1$. This introduces a Lagrange multiplier $p \equiv p(\mathbf{X})$, which must be computed together with the deformation.

The material law becomes, for an isotropic material

$$W(C) = W^{\mathrm{mat}}(I_1, I_2) - \frac{p}{2}(I_3 - 1)$$

This gives: $T_{MN} = 2\frac{\partial W^{\mathrm{mat}}}{\partial C_{MN}} - p\left(C^{-1}\right)_{MN}$, or equivalently

$$\sigma_{ij} = \sigma_{ij}^{\mathrm{mat}} - p\delta_{ij}$$

Incompressible strain energy functions

Recall that for incompressible materials we have the constraint

$$\det F = 1$$

(everywhere), i.e. $I_3 = 1$. This introduces a Lagrange multiplier $p \equiv p(\mathbf{X})$, which must be computed together with the deformation.

The material law becomes, for an isotropic material

$$W(C) = W^{\text{mat}}(I_1, I_2) - \frac{p}{2}(I_3 - 1)$$

This gives: $T_{MN} = 2\frac{\partial W^{\text{mat}}}{\partial C_{MN}} - p\left(C^{-1}\right)_{MN}$, or equivalently

$$\sigma_{ij} = \sigma_{ij}^{\text{mat}} - p\delta_{ij}$$

Example strain energy functions

Incompressible strain energies:

- Neo-Hookean: $W^{\mathrm{mat}}(I_1, I_2) = c_1(I_1 - 3)$
- Mooney-Rivlin: $W^{\mathrm{mat}}(I_1, I_2) = c_1(I_1 - 3) + c_2(I_2 - 3)$
- Veronda-Westman: $W^{\mathrm{mat}}(I_1, I_2) = c_1 e^{\alpha(I_1 - 3)} + c_2(I_2 - 3)$
- Similar exponential laws are often used in biology

$$W^{\mathrm{mat}}(C) = c_1 e^{\alpha(Q(C)-1)}$$

where $Q(C)$ is a quadratic in the entries of C

A compressible strain energy: the compressible Neo-Hookean law

$$W(I_1, I_2, I_3) = c_1(\bar{I}_1 - 3) + c_3(J - 1)^2$$

Material law for (compressible) linear elasticity

Stress σ_{ij} is linearly related to strain ϵ_{ij}:

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl}$$

For *isotropic materials*, it can be shown that this relationship must be of the form

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\delta_{ij}\epsilon_{kk}$$

where material parameters λ and μ are the *Lamé coefficients*

This relationship is often re-written using derived parameters E (Young's modulus) and ν (Poisson's ratio)

Material law for (compressible) linear elasticity

Stress σ_{ij} is linearly related to strain ϵ_{ij}:

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl}$$

For *isotropic materials*, it can be shown that this relationship must be of the form

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\delta_{ij}\epsilon_{kk}$$

where material parameters λ and μ are the *Lamé coefficients*

This relationship is often re-written using derived parameters E (Young's modulus) and ν (Poisson's ratio)

Material law for (compressible) linear elasticity

Stress σ_{ij} is linearly related to strain ϵ_{ij}:

$$\sigma_{ij} = C_{ijkl}\epsilon_{kl}$$

For *isotropic materials*, it can be shown that this relationship must be of the form

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\delta_{ij}\epsilon_{kk}$$

where material parameters λ and μ are the *Lamé coefficients*

This relationship is often re-written using derived parameters E (Young's modulus) and ν (Poisson's ratio)

Overall governing equations:

Overall governing equations: static, incompressible nonlinear elasticity

Given a material relationship $W \equiv W(C,p)$, $T \equiv 2\frac{\partial W}{\partial C}$:

Find $\mathbf{x} \equiv \mathbf{x}(\mathbf{X})$ and $p \equiv p(\mathbf{X})$ satisfying

$$\begin{aligned} \frac{\partial}{\partial X_M}\left(T_{MN}(\mathbf{x},p)\frac{\partial x_i}{\partial X_M}\right) + \rho_0 b_i &= 0 \\ \det F(\mathbf{x}) &= 1 \end{aligned}$$

with boundary conditions:

$$\begin{aligned} \mathbf{x} &= \mathbf{x}^* && \text{on } \Gamma_1 \\ TF^{\mathrm{T}}\mathbf{N} &= \mathbf{s} && \text{on } \Gamma_2 \end{aligned}$$

Overall governing equations: static, compressible linear elasticity

Given the material relationship

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\delta_{ij}\epsilon_{kk}$$

find $\mathbf{u} \equiv \mathbf{u}(\mathbf{X})$ satisfying

$$\frac{\partial\sigma_{ij}(\mathbf{u})}{\partial X_j} + \rho_0 b_i = 0$$

with boundary conditions:

$$\begin{aligned} \mathbf{u} &= \mathbf{u}^* \qquad \text{on } \Gamma_1 \\ \sigma\mathbf{n} &= \mathbf{s} \qquad \text{on } \Gamma_2 \end{aligned}$$

Sometimes this is expanded and expressed explicitly in terms of $\mathbf{u}$

$$(\lambda+\mu)\nabla(\nabla\cdot\mathbf{u}) + \mu\nabla^2\mathbf{u} + \rho_0\mathbf{b} = 0$$

Overall governing equations: static, compressible linear elasticity

Given the material relationship

$$\sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\delta_{ij}\epsilon_{kk}$$

find $\mathbf{u} \equiv \mathbf{u}(\mathbf{X})$ satisfying

$$\frac{\partial\sigma_{ij}(\mathbf{u})}{\partial X_j} + \rho_0 b_i = 0$$

with boundary conditions:

$$\begin{aligned} \mathbf{u} &= \mathbf{u}^* && \text{on } \Gamma_1 \\ \sigma\mathbf{n} &= \mathbf{s} && \text{on } \Gamma_2 \end{aligned}$$

Sometimes this is expanded and expressed explicitly in terms of $\mathbf{u}$

$$(\lambda + \mu)\nabla(\nabla \cdot \mathbf{u}) + \mu\nabla^2\mathbf{u} + \rho_0\mathbf{b} = 0$$

Overall governing equations: fluids

For fluids, $\mathbf{u}$ is used to denote *flow rather than displacement* and $\mathbf{x}$ is the independent variable (i.e. an Eulerian formulation is used).

For incompressible flow:

- **Kinematics:** Incompressibility $\Rightarrow \nabla \cdot \mathbf{u} = 0$ again, ...
- **Balance law:** use time-dependent Eulerian, i.e. $\rho \frac{D}{Dt} u_i = \frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i$
- **Material law:** Stress is a function of strain-rate, one material parameter, μ, the viscosity, and of pressure, as before

Overall, the Navier-Stokes equations are: find $\mathbf{u}$ and p satisfying

$$\begin{aligned} \rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) &= -\nabla p + \mu\nabla^2\mathbf{u} + \rho\mathbf{b} \\ \nabla\cdot\mathbf{u} &= 0 \end{aligned}$$

Overall governing equations: fluids

For fluids, $\mathbf{u}$ is used to denote *flow rather than displacement* and $\mathbf{x}$ is the independent variable (i.e. an Eulerian formulation is used).

For incompressible flow:

- **Kinematics:** Incompressibility $\Rightarrow \nabla \cdot \mathbf{u} = 0$ again, ...
- **Balance law**: use time-dependent Eulerian, i.e. $\rho \frac{D}{Dt} u_i = \frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i$
- **Material law**: Stress is a function of *strain-rate*, one material parameter, μ, the viscosity; and of pressure, as before

Overall, the Navier-Stokes equations are: find $\mathbf{u}$ and p satisfying

$$\begin{aligned} \rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}\right) &= -\nabla p + \mu \nabla^2 \mathbf{u} + \rho \mathbf{b} \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned}$$

Overall governing equations: fluids

For fluids, $\mathbf{u}$ is used to denote *flow rather than displacement* and $\mathbf{x}$ is the independent variable (i.e. an Eulerian formulation is used).

For incompressible flow:

- **Kinematics:** Incompressibility $\Rightarrow \nabla \cdot \mathbf{u} = 0$ again, ...
- **Balance law**: use time-dependent Eulerian, i.e. $\rho \frac{D}{Dt} u_i = \frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i$
- **Material law**: Stress is a function of *strain-rate*, one material parameter, μ, the viscosity; and of pressure, as before

Overall, the Navier-Stokes equations are: find $\mathbf{u}$ and p satisfying

$$\begin{aligned}\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) &= -\nabla p + \mu\nabla^2\mathbf{u} + \rho\mathbf{b} \\ \nabla\cdot\mathbf{u} &= 0\end{aligned}$$

Overall governing equations: fluids

For fluids, $\mathbf{u}$ is used to denote *flow rather than displacement* and $\mathbf{x}$ is the independent variable (i.e. an Eulerian formulation is used).

For incompressible flow:

- **Kinematics:** Incompressibility $\Rightarrow \nabla \cdot \mathbf{u} = 0$ again, ...
- **Balance law**: use time-dependent Eulerian, i.e. $\rho \frac{D}{Dt} u_i = \frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i$
- **Material law**: Stress is a function of *strain-rate*, one material parameter, μ, the viscosity; and of pressure, as before

Overall, the Navier-Stokes equations are: find $\mathbf{u}$ and p satisfying

$$\begin{aligned} \rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) &= -\nabla p + \mu\nabla^2\mathbf{u} + \rho\mathbf{b} \\ \nabla\cdot\mathbf{u} &= 0 \end{aligned}$$

Overall governing equations: fluids

For fluids, $\mathbf{u}$ is used to denote *flow rather than displacement* and $\mathbf{x}$ is the independent variable (i.e. an Eulerian formulation is used).

For incompressible flow:

- **Kinematics:** Incompressibility $\Rightarrow \nabla \cdot \mathbf{u} = 0$ again, ...
- **Balance law**: use time-dependent Eulerian, i.e. $\rho \frac{D}{Dt} u_i = \frac{\partial \sigma_{ij}}{\partial x_j} + \rho b_i$
- **Material law**: Stress is a function of *strain-rate*, one material parameter, μ, the viscosity; and of pressure, as before

Overall, the Navier-Stokes equations are: find $\mathbf{u}$ and p satisfying

$$\begin{aligned} \rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\boldsymbol{\nabla}\mathbf{u}\right) &= -\boldsymbol{\nabla} p + \mu\nabla^2\mathbf{u} + \rho\mathbf{b} \\ \nabla\cdot\mathbf{u} &= 0 \end{aligned}$$

Numerical methods (incompressible nonlinear elasticity only)

Weak form

We cam compute the weak form as before

- multiplying (inner product) of the first equation with a test function $\mathbf{v} \in \mathcal{V}$, integrate, use divergence theorem.
- multiply second equation with $q \in \mathcal{W}$, integrate

Find $\mathbf{x} \in \mathcal{V}$ and $p \in \mathcal{W}$ such $\mathbf{x} = \mathbf{x}^*$ on Γ_1 and

$$\begin{aligned} \int_{\Omega_0} T_{MN}(\mathbf{x}, p) \frac{\partial x_i}{\partial X_N} \frac{\partial v_i}{\partial X_M} \, \mathrm{d}V_0 - \int_{\Omega_0} \rho_0 \mathbf{b} \cdot \mathbf{v} \, \mathrm{d}V_0 - \int_{\Gamma_2} \mathbf{s} \cdot \mathbf{v} \, \mathrm{d}S_0 & \\ + \int_{\Omega_0} q \, (\det F(\mathbf{x}) - 1) \, \mathrm{d}V_0 &= 0 \\ & \forall \mathbf{v} \in \mathcal{V}_0, q \in \mathcal{W} \end{aligned}$$

Newton's method

Using $\mathbf{u}$ as the unknown instead of $\mathbf{x}$, write weak problem as:

Find $\mathbf{u}^h \in \mathcal{V}^h$, $p^h \in \mathcal{W}^h$ such that $\mathbf{u} = \mathbf{u}^*$ on Γ_1 and:

$$\mathcal{F}(\mathbf{u}^h, p^h, \mathbf{v}, q) = 0 \qquad \forall \mathbf{v} \in \mathcal{V}_0{}^h, q \in \mathcal{W}^h$$

Use **quadratic basis functions for displacement, linear for pressure**. This is necessary for a 'stable' scheme (accuracy).

Suppose there are

- N quadratic bases, $\phi_1, \dots, \phi_N$
- M linear bases, $\psi_1, \dots, \psi_M$:

Let $\mathbf{v} = \begin{bmatrix} \phi_i \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \phi_i \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ \phi_i \end{bmatrix}$ and $q = \psi_i \qquad \Rightarrow 3N + M$ nonlinear eqns

Solve using Newton's method as described in lecture 4.

Newton's method

Using $\mathbf{u}$ as the unknown instead of $\mathbf{x}$, write weak problem as:

Find $\mathbf{u}^h \in \mathcal{V}^h$, $p^h \in \mathcal{W}^h$ such that $\mathbf{u} = \mathbf{u}^*$ on Γ_1 and:

$$\mathcal{F}(\mathbf{u}^h, p^h, \mathbf{v}, q) = 0 \qquad \forall \mathbf{v} \in {\mathcal{V}_0}^h, q \in \mathcal{W}^h$$

Use **quadratic basis functions for displacement, linear for pressure**. This is necessary for a 'stable' scheme (accuracy).

Suppose there are

- N quadratic bases, $\phi_1, \dots, \phi_N$
- M linear bases, $\psi_1, \dots, \psi_M$:

Let $\mathbf{v} = \begin{bmatrix} \phi_i \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \phi_i \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ \phi_i \end{bmatrix}$ and $q = \psi_i \qquad \Rightarrow 3N + M$ nonlinear eqns

Solve using Newton's method as described in lecture 4.

Newton's method

Using $\mathbf{u}$ as the unknown instead of $\mathbf{x}$, write weak problem as:

Find $\mathbf{u}^h \in \mathcal{V}^h$, $p^h \in \mathcal{W}^h$ such that $\mathbf{u} = \mathbf{u}^*$ on Γ_1 and:

$$\mathcal{F}(\mathbf{u}^h, p^h, \mathbf{v}, q) = 0 \qquad \forall \mathbf{v} \in {\mathcal{V}_0}^h, q \in \mathcal{W}^h$$

Use **quadratic basis functions for displacement, linear for pressure**. This is necessary for a 'stable' scheme (accuracy).

Suppose there are

- N quadratic bases, $\phi_1, \dots, \phi_N$
- M linear bases, $\psi_1, \dots, \psi_M$:

Let $\mathbf{v} = \begin{bmatrix} \phi_i \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \phi_i \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ \phi_i \end{bmatrix}$ and $q = \psi_i \qquad \Rightarrow 3N + M$ nonlinear eqns

Solve using Newton's method as described in lecture 4.

Newton's method

Using $\mathbf{u}$ as the unknown instead of $\mathbf{x}$, write weak problem as:

Find $\mathbf{u}^h \in \mathcal{V}^h$, $p^h \in \mathcal{W}^h$ such that $\mathbf{u} = \mathbf{u}^*$ on Γ_1 and:

$$\mathcal{F}(\mathbf{u}^h, p^h, \mathbf{v}, q) = 0 \qquad \forall \mathbf{v} \in \mathcal{V}_0{}^h, q \in \mathcal{W}^h$$

Use **quadratic basis functions for displacement, linear for pressure**. This is necessary for a 'stable' scheme (accuracy).

Suppose there are

- N quadratic bases, $\phi_1, \ldots, \phi_N$
- M linear bases, $\psi_1, \ldots, \psi_M$:

Let $\mathbf{v} = \begin{bmatrix} \phi_i \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \phi_i \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ \phi_i \end{bmatrix}$ and $q = \psi_i$ $\quad \Rightarrow 3N + M$ nonlinear eqns

Solve using Newton's method as described in lecture 4.

Newton's method

Using $\mathbf{u}$ as the unknown instead of $\mathbf{x}$, write weak problem as:

Find $\mathbf{u}^h \in \mathcal{V}^h$, $p^h \in \mathcal{W}^h$ such that $\mathbf{u} = \mathbf{u}^*$ on Γ_1 and:

$$\mathcal{F}(\mathbf{u}^h, p^h, \mathbf{v}, q) = 0 \qquad \forall \mathbf{v} \in {\mathcal{V}_0}^h, q \in \mathcal{W}^h$$

Use **quadratic basis functions for displacement, linear for pressure**. This is necessary for a 'stable' scheme (accuracy).

Suppose there are

- N quadratic bases, $\phi_1, \dots, \phi_N$
- M linear bases, $\psi_1, \dots, \psi_M$:

Let $\mathbf{v} = \begin{bmatrix} \phi_i \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \phi_i \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ \phi_i \end{bmatrix}$ and $q = \psi_i \quad \Rightarrow 3N + M$ nonlinear eqns

Solve using Newton's method as described in lecture 4.

Newton's method

Using $\mathbf{u}$ as the unknown instead of $\mathbf{x}$, write weak problem as:

Find $\mathbf{u}^h \in \mathcal{V}^h$, $p^h \in \mathcal{W}^h$ such that $\mathbf{u} = \mathbf{u}^*$ on Γ_1 and:

$$\mathcal{F}(\mathbf{u}^h, p^h, \mathbf{v}, q) = 0 \qquad \forall \mathbf{v} \in {\mathcal{V}_0}^h, q \in \mathcal{W}^h$$

Use **quadratic basis functions for displacement, linear for pressure**. This is necessary for a 'stable' scheme (accuracy).

Suppose there are

- N quadratic bases, $\phi_1, \ldots, \phi_N$
- M linear bases, $\psi_1, \ldots, \psi_M$:

Let $\mathbf{v} = \begin{bmatrix} \phi_i \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \phi_i \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ \phi_i \end{bmatrix}$ and $q = \psi_i \quad \Rightarrow 3N + M$ nonlinear eqns

Solve using Newton's method as described in lecture 4.

Object-oriented design

```
SolidMechanicsProblemDefinition
    mBodyForce
    mFixedNodes
    mFixedNodeLocations
    mNeumannBoundaryElements
    mTractions
```

Chaste

Object-oriented design

```
IncompressibleNonlinearElasticitySolver
    Solve(mesh, solidMechProblemDefn, absIncompMaterialLaw)
```

▷ *Use Newton's method to solve the given problem*

Object-oriented design

`AbstractIncompressibleMaterialLaw`

Abs. method: `GetStrainEnergyValue(C)`

▷ *Take in C, return $W(C)$*

This doesn't work as code needs to use $T = 2\frac{\partial W}{\partial C}$ (and also $\frac{\partial^2 W}{\partial C_{MN} \partial C_{PQ}}$)

Object-oriented design

`AbstractIncompressibleMaterialLaw`

Abs. method: `ComputeStressAndStressDerivative(C,p)`

- *Take in C, p, return* $2\frac{\partial W}{\partial C}$ *and* $4\frac{\partial^2 W}{\partial C_{MN}\partial C_{PQ}}$

`AbstractIsotropicIncompressibleMaterialLaw` *inherits from* `(above)`:

Method:	`ComputeStressAndStressDerivative(C,p)`	
Abs. method:	`Get_dW_dI1(I1,I2)`	Get $\frac{\partial W}{\partial I_1}$
Abs. method:	`Get_dW_dI2(I1,I2)`	Get $\frac{\partial W}{\partial I_2}$
Abs. method:	`Get_d2W_dI1(I1,I2)`	Get $\frac{\partial^2 W}{\partial {I_1}^2}$
Abs. method:	`Get_d2W_dI2(I1,I2)`	
Abs. method:	`Get_d2W_dI1dI2(I1,I2)`	

Object-oriented design

The Mooney-Rivlin law is

$$W^{\mathrm{mat}}(I_1, I_2) = c_1(I_1 - 3) + c_2(I_2 - 3)$$

`MooneyRivlinMaterialLaw` *inherits from* `AbsIsotropicIncompMaterialLaw`:

```
    Implemented method: Get_dW_dI1(I1,I2)
        ▷ return $c_1$
    Implemented method: Get_dW_dI2(I1,I2)
        ▷ return $c_2$
    Implemented method: Get_d2W_dI1(I1,I2)
        ▷ return 0, etc.
```