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 $\mathbf{y}^0 (= \mathbf{y}_0), \mathbf{y}^1, \mathbf{y}^2, \ldots$ 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.

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 $[0, T_{\text{end}}]$. Let $T_{\text{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\text{end}}\Delta t) = \mathcal{O}(\Delta t)$, 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 $\frac{dy}{dt} = f(t, y)$

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 $[0, T_{\text{end}}]$. Let $T_{\text{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\text{end}}\Delta t) = \mathcal{O}(\Delta t)$, 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 $[0, T_{\text{end}}]$. Let $T_{\text{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\text{end}}\Delta t) = \mathcal{O}(\Delta t)$, 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 $[0, T_{\text{end}}]$. Let $T_{\text{end}} = N\Delta t$, i.e. let N be the total number timesteps taken.

For the Euler methods we expect $e_N = \mathcal{O}(N\Delta t^2) = \mathcal{O}(T_{\text{end}}\Delta t) = \mathcal{O}(\Delta t)$, 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

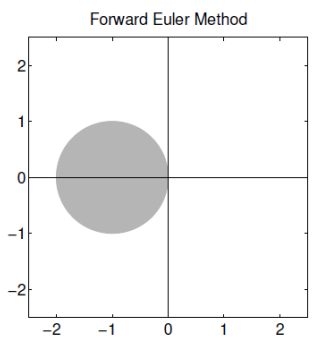
$$\frac{\mathrm{d}y}{\mathrm{d}t} = \lambda y, \text{ with } y(0) = 1 \qquad \Rightarrow \qquad y = e^{\lambda t}$$

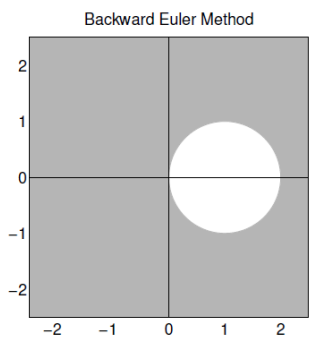
If $\lambda < 0$, then $y \to 0$ as $t \to \infty$.

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**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\mathcal{O}(\Delta t)$
- **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**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\mathcal{O}(\Delta t)$
- **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**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\mathcal{O}(\Delta t)$
- **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**: $y^{n+1} = y^n + \Delta t\, f(t^n, y^n)$ $\mathcal{O}(\Delta t)$
- **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**: $y^{n+2} = y^{n+1} + \frac{1}{2}\Delta t \left(3f^{n+1} - f^n\right)$ $\mathcal{O}(\Delta t^2)$

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* $\mathbf{y}$

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` ←

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`).