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

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>
```

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

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

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

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

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

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

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

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

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

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

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

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

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 $U_1, \ldots, U_N$ of $u_h = \sum U_i \phi_i$ such that

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

This is a general N-dimensional nonlinear system.

An iterative approach is required to solve nonlinear systems. Let u_h^k (equivalently, $\mathbf{U^k} = [U_1^k, \ldots, U_N^k]$) 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 $U_1, \ldots, U_N$ of $u_h = \sum U_i \phi_i$ such that

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

This is a general N-dimensional nonlinear system.

An iterative approach is required to solve nonlinear systems. Let u_h^k (equivalently, $\mathbf{U}^k = [U_1^k, \ldots, U_N^k]$) 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 $U_1, \ldots, U_N$ of $u_h = \sum U_i \phi_i$ such that

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

This is a general N-dimensional nonlinear system.

An iterative approach is required to solve nonlinear systems. Let u_h^k (equivalently, $\mathbf{U^k} = [U_1^k, \ldots, U_N^k]$) 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). 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:

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

(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:

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

(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:

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

(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:

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

(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. $\mathbf{u}$ *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.

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

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

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

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:

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

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

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

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

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

where

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

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`[1] *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)`

[1] Slightly simplified

Cell models in Chaste

`AbstractOdeSystem`

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

`AbstractCardiacCell`[1] *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)`

[1] Slightly simplified

Cell models in Chaste

`AbstractOdeSystem`

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

`AbstractCardiacCell`[1] *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)`

[1] 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.*

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.