Implementing the Finite Element Method

Part II: Global Assembly and Linear Solvers

Colin Cotter, David Ham, **Stephan Kramer**

`stephan.kramer@imperial.ac.uk`

Applied Modelling and Computation Group (AMCG),

Faculty of Earth Science and Engineering,

Imperial College London.

Imperial College London

Global Assembly

Imperial College London

Back to our favourite equation

$$-\nabla^2\psi = f(\vec{x})$$

on some domain Ω with boundary condition $\nabla\psi \cdot \mathbf{n} = 0$.

Back to our favourite equation

$$-\nabla^2\psi = f(\vec{x})$$

on some domain Ω with boundary condition $\nabla\psi \cdot \mathbf{n} = 0$.

Multiply by a test function N and integrate:

$$-\int_\Omega N\nabla^2\psi dV = \int_\Omega N f(\vec{x}) dV$$

Back to our favourite equation

$$-\nabla^2\psi = f(\vec{x})$$

on some domain Ω with boundary condition $\nabla\psi \cdot \mathbf{n} = 0$.

Multiply by a test function N and integrate:

$$-\int_\Omega N\nabla^2\psi dV = \int_\Omega N f(\vec{x})dV$$

Integrate by parts.

$$\int_\Omega \nabla N \nabla\psi dV - \int_\Gamma N\nabla\psi \cdot \mathbf{n} dA = \int_\Omega N f(\vec{x})dV$$

Imperial College
London

Back to our favourite equation

$$-\nabla^2\psi = f(\vec{x})$$

on some domain Ω with boundary condition $\nabla\psi \cdot \mathbf{n} = 0$.

Multiply by a test function N and integrate:

$$-\int_\Omega N\nabla^2\psi dV = \int_\Omega N f(\vec{x}) dV$$

Integrate by parts.

$$\int_\Omega \nabla N \nabla\psi dV - \int_\Gamma N\nabla\psi \cdot \mathbf{n} dA = \int_\Omega N f(\vec{x}) dV$$

Back to our favourite equation

$$-\nabla^2\psi = f(\vec{x})$$

on some domain Ω with boundary condition $\nabla\psi \cdot \mathbf{n} = 0$.

Multiply by a test function N and integrate:

$$-\int_\Omega N\nabla^2\psi dV = \int_\Omega N f(\vec{x}) dV$$

Integrate by parts.

$$\int_\Omega \nabla N \nabla\psi dV = \int_\Omega N f(\vec{x}) dV$$

Discretisation

$$\int_\Omega \vec{\nabla} N \cdot \vec{\nabla} \psi dV = \int_\Omega N f(\vec{x}) dV$$

Discretisation follows by choosing a set of tests function N_i and decomposing our solution in a number of trial functions (in this case the same) N_j:

$$\psi(x) = \sum_j \psi_j N_j(x)$$

which gives:

$$\sum_j \int_\Omega \vec{\nabla} N_i \cdot \vec{\nabla} N_j \psi_j dV = \int_\Omega N_i f(\vec{x}) dV$$

Imperial College
London

Or in matrix form

$$\sum_j \int_\Omega \vec{\nabla} N_i \cdot \vec{\nabla} N_j \psi_j dV = \int_\Omega N_i f(\vec{x}) dV$$

can be written in matrix form:

$$\sum_j A_{ij}\psi_j = b_i$$

where

$$A_{ij} = \int_\Omega \vec{\nabla} N_i \cdot \vec{\nabla} N_j dV$$

$$b_i = \int_\Omega N_i f(\vec{x}) dV$$

Matrix form

$$
\begin{pmatrix}
A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & \cdots \\
A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & \cdots \\
A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & \cdots \\
A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & \cdots \\
A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{pmatrix}
\begin{pmatrix}
\psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \\ \psi_5 \\ \vdots
\end{pmatrix}
=
\begin{pmatrix}
b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \vdots
\end{pmatrix}
$$

$$A_{ij} = \int_\Omega \vec{\nabla} N_i \cdot \vec{\nabla} N_j dV$$

$$b_j = \int_\Omega N_i f(\vec{x}) dV$$

Matrix form

$$\begin{pmatrix} A_{11} & A_{12} & 0 & 0 & 0 & \dots \\ A_{21} & A_{22} & A_{23} & 0 & 0 & \dots \\ 0 & A_{32} & A_{33} & A_{34} & 0 & \dots \\ 0 & 0 & A_{43} & A_{44} & A_{45} & \dots \\ 0 & 0 & 0 & A_{54} & A_{55} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \\ \psi_5 \\ \vdots \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \vdots \end{pmatrix}$$

$$A_{ij} = \int_\Omega \vec{\nabla} N_i \cdot \vec{\nabla} N_j dV$$

$$b_j = \int_\Omega N_i f(\vec{x}) dV$$

Most matrix entries are zero! (example for 1D mesh)

Imperial College London

Matrix form

$$\begin{pmatrix} A_{11} & A_{12} & 0 & 0 & 0 & \dots \\ A_{21} & A_{22} & A_{23} & 0 & 0 & \dots \\ 0 & A_{32} & A_{33} & A_{34} & 0 & \dots \\ 0 & 0 & A_{43} & A_{44} & A_{45} & \dots \\ 0 & 0 & 0 & A_{54} & A_{55} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \\ \psi_5 \\ \vdots \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ \vdots \end{pmatrix}$$

$$A_{ij} = \int_\Omega \vec{\nabla} N_i \cdot \vec{\nabla} N_j dV, \quad b_j = \int_\Omega N_i f(\vec{x}) dV$$

More generally: $A_{ij} \neq 0$ if and only if the nodes i and j belong to the same element.

Imperial College London

CSR matrices

$$\begin{pmatrix} \mathbf{1.0} & \mathbf{4.0} & 0.0 & 0.0 & 0.0 & 0.0 \\ \mathbf{2.0} & \mathbf{1.0} & 0.0 & \mathbf{3.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{2.0} & 0.0 & 0.0 & 0.0 \\ 0.0 & \mathbf{5.0} & 0.0 & \mathbf{2.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & \mathbf{2.0} & \mathbf{1.0} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{7.0} \end{pmatrix}$$

For *sparse* matrices it is more efficient to only store the nonzero entries.

Compressed Sparse Row (CSR):

values: $1.0 \quad 4.0 \,\big|\, 2.0 \quad 1.0 \quad 3.0 \,\big|\, 2.0 \,\big|\, 5.0 \quad 2.0 \,\big|\, 2.0 \quad 1.0 \,\big|\, 7.0$

CSR matrices

$$\begin{pmatrix} \mathbf{1.0} & \mathbf{4.0} & 0.0 & 0.0 & 0.0 & 0.0 \\ \mathbf{2.0} & \mathbf{1.0} & 0.0 & \mathbf{3.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{2.0} & 0.0 & 0.0 & 0.0 \\ 0.0 & \mathbf{5.0} & 0.0 & \mathbf{2.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & \mathbf{2.0} & \mathbf{1.0} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{7.0} \end{pmatrix}$$

For *sparse* matrices it is more efficient to only store the nonzero entries.

Compressed Sparse Row (CSR):

values:	1.0	4.0	2.0	1.0	3.0	2.0	5.0	2.0	2.0	1.0	7.0
columns:	1	2	1	2	4	3	2	4	4	5	6

CSR matrices

$$\begin{pmatrix} \mathbf{1.0} & \mathbf{4.0} & 0.0 & 0.0 & 0.0 & 0.0 \\ \mathbf{2.0} & \mathbf{1.0} & 0.0 & \mathbf{3.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{2.0} & 0.0 & 0.0 & 0.0 \\ 0.0 & \mathbf{5.0} & 0.0 & \mathbf{2.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & \mathbf{2.0} & \mathbf{1.0} & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \mathbf{7.0} \end{pmatrix}$$

For *sparse* matrices it is more efficient to only store the nonzero entries.

Compressed Sparse Row (CSR):

values:	1.0	4.0	2.0	1.0	3.0	2.0	5.0	2.0	2.0	1.0	7.0
columns:	1	2	1	2	4	3	2	4	4	5	6

row start: 1 3 6 7 9 10

Back to the code: sparsities

Usually multiple matrices will have the same non-zero structure. This structure is therefore stored as a separate object called csr_sparsity.

```
use sparse_tools
type(csr_sparsity):: sparsity
integer:: rows, columns, entries

! Number of matrix rows:
rows = 100
! Number of matrix columns:
columns = 100
! Number of non-zero entries in the sparsity:
entries = 300
call allocate(sparsity, rows, columns, entries, name="MySparsity")
```

Back to the code: sparsities

In a lot of cases the sparsity of the matrix of a FEM discretisation is defined by:

> $A_{ij} \neq 0$ if and only if the nodes i and j belong to the same (at least one) element.

i.e. in an expression like:

$$\int \nabla N_i \cdot \nabla M_j$$

both the test function N_i and the trial function M_j overlap in at least one element. In this case we can use make_sparsity().

```
use fields
use sparse_tools
type(mesh_type):: test_mesh, trial_mesh
type(csr_sparsity):: sparsity

! create a sparsity based on test_mesh and trial_mesh
sparsity=make_sparsity(test_mesh, trial_mesh, name="MySparsity")
```

CSR matrices

Now we have a sparsity we can make a matrix

```
use fields
use sparse_tools
type(mesh_type):: test_mesh, trial_mesh
type(csr_sparsity):: sparsity
type(csr_matrix):: A

! create a sparsity based on test_mesh and trial_mesh
sparsity=make_sparsity(test_mesh, trial_mesh, name="MySparsity")
! allocate a matrix with this sparsity
call allocate(A, sparsity, name="MyMatrix")
```

First thing to do is zero all entries:

```
! zero all entries
call zero(A)
```

Setting values in the matrix

Then we would like to set the value of some entries:

```fortran
! set A_12=pi
call set(A, 1, 2, 3.14159)
```

or add something to previously set entries:

```fortran
! add 2.0 to A_{10,12}
call addto(A, 10, 12, 2.0)
```

You can also addto/set multiple entries at once:

```fortran
real, dimension(1:2,1:2):: val_mat

! add val_mat to a (non-contigous) submatrix of A
call addto(A, (/ 2,5 /), (/ 2,5 /), val_mat)
```

Setting multiple values in the matrix

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & \mathbf{1.0} & 0.0 & 0.0 & 0.0 & \mathbf{2.0} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & \mathbf{3.0} & 0.0 & 0.0 & 0.0 & \mathbf{4.0} \end{pmatrix}$$

```fortran
real, dimension(1:2,1:2):: val_mat = &
   reshape( (/ 1.0, 2.0, 3.0, 4.0 /), (/ 2, 2 /) )

call zero(A)
! add val_mat to a (non-contigous) submatrix of A
call addto(A, (/ 2,5 /), (/ 2,5 /), val_mat)
```

Element-wise matrix assembly

This is handy for setting all coefficients related to the integrals inside one element. Let's consider again

$$A_{ij} = \int_\Omega \nabla N_i \cdot \nabla N_j \, dV = \sum_e \int_{\Omega_e} \nabla N_i \cdot \nabla N_j \, dV$$

```
type(scalar_field):: psi
real, dimension(1:ele_loc(psi,ele), 1:ele_loc(psi,ele)):: ele_mat
integer, dimension(:), pointer:: ele_psi

! compute ele_mat=\int dN_i dN_j for element ele
...
! return a pointer to the node numbers of element ele
ele_psi => ele_nodes(psi, ele)
! add ele_mat into (non-contigous) submatrix of A
call addto(A, ele_psi, ele_psi, ele_mat)
```

Element-wise matrix assembly

Similarly the rhs is added in element by element:

$$b_i = \int_\Omega N_i f(\mathbf{x}) dV = \sum_e \int_{\Omega_e} N_i f(\mathbf{x}) dV$$

```
type(scalar_field):: psi, rhs
real, dimension(1:ele_loc(psi,ele), 1:ele_loc(psi,ele)):: ele_mat
real, dimension(1:ele_loc(psi,ele)):: ele_rhs
integer, dimension(:), pointer:: ele_psi

! compute ele_mat=\int dN_i dN_j for element ele
! and ele_rhs=\int N_i f for element ele
...
! return a pointer to the node numbers of element ele
ele_psi => ele_nodes(psi, ele)
! add ele_mat into (non-contigous) submatrix of A
call addto(A, ele_psi, ele_psi, ele_mat)
! add ele_rhs to the rhs of the equation
call addto(rhs, ele_psi, ele_rhs)
```

The assembly is done

```
! Assemble A element by element.
do ele=1, element_count(psi)
   call assemble_element_contribution(A, rhs, positions, psi, &
     rhs_func, ele)
end do
```

The assembly is done (almost)

```fortran
! Assemble A element by element.
do ele=1, element_count(psi)
   call assemble_element_contribution(A, rhs, positions, psi, &
     rhs_func, ele)
end do

! It is necessary to fix the value of one node in the solution.
! We choose node 1.
call set(A, 1, 1, INFINITY)
```

We set a huge number on the diagonal to fix ψ_1:

$$\sum_j A_{1j}\psi_j = A_{11}\psi_1 + \sum_{j\neq 1} A_{1j}\psi_j = b_1$$

$$A_{11}\psi_1 \approx b_1 \implies \psi_1 \approx 0.0$$

Solving the equation

PETSc

PETSc, the Portable, Extensible Toolkit for Scientific Computation - library with a.o. a large collection of linear solvers, preconditioners, etc.

- Range of avalailable matrix formats, linear solvers and preconditioners.
- Provides common interface, also to yet other libraries:
  - Hypre: BoomerAMG
  - Prometheus
  - Trilinos/ML
- Interface in Fortran, C, C++ and Python(!).
- Has already been used in very many large-scale applications.

Linear solvers

Solution of a linear system

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

Direct methods

Construct inverse A^{-1} of matrix and so computes $\mathbf{x} = A^{-1}\mathbf{b}$. Construction of dense inverse matrix is expensive in both memory and time.

Iterative methods

Series of approximations $\mathbf{x}^k$ with improvement each step, so that (hopefully) $\mathbf{x}^k$ converges to $\mathbf{x}$.

Residual

Solution of a linear system

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

Approximation x^k in k-th iteration.

Error: $\mathbf{e}^k = \mathbf{x}^k - \mathbf{x}$ (hopefully $\mathbf{e}^k \to 0$)

Residual: $\mathbf{r}^k = A\mathbf{x}^k - \mathbf{b}$

Note: $A\mathbf{e}^k = A\mathbf{x}^k - A\mathbf{x} = \mathbf{r}^k$

Stationary iterative method

Suppose M is an approximation of the matrix A such that M^{-1} is easy to calculate.

Then the error can be approximated by

$$\mathbf{e}^k = \mathbf{x}^k - \mathbf{x} \approx M^{-1} A \mathbf{e}^k = M^{-1} \mathbf{r}^k$$

Stationary iterative method:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - M^{-1} \mathbf{r}^k$$

Jacobi and Gauss Seidel

Jacobi iteration

Take approximate matrix M to be only the diagonal of A.

Gauss Seidel iteration

Take M to be everything on or below the diagonal:

$$\begin{pmatrix} A_{11} & 0 & 0 & \dots \\ A_{21} & A_{22} & 0 & \dots \\ A_{31} & A_{32} & A_{33} & \dots \\ \dots & \dots & \dots & \ddots \end{pmatrix} \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ \dots \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \dots \end{pmatrix}$$

thus computing error approximation $\mathbf{z}^k = M^{-1}\mathbf{r}^k$.

Imperial College London

Krylov subspace methods

Consider iteration:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k \mathbf{r}^k$$
$$\implies \mathbf{r}^{k+1} = \mathbf{r}^k + \alpha_k A \mathbf{r}^k$$

Working out:

$$\mathbf{r}^0 = \mathbf{b} - A\mathbf{x}^0$$
$$\mathbf{r}^1 = \mathbf{r}^0 + \alpha_0 A \mathbf{r}^0$$
$$\mathbf{r}^2 = \mathbf{r}^1 + \alpha_1 A \mathbf{r}^1 = \mathbf{r}^0 + (\alpha_0 + \alpha_1) A \mathbf{r}^0 + \alpha_1 A^2 \mathbf{r}^0$$
$$\dots$$

Thus $\mathbf{r}^k$ is linear combination of $\mathbf{r}^0, A\mathbf{r}^0, A^2\mathbf{r}^0, \dots A^k\mathbf{r}^0$.

Krylov subspace methods

Krylov subspace is linear space spanned by these vectors:

$$K^k = span\left(\mathbf{r}^0, A\mathbf{r}^0, A^2\mathbf{r}^0, \ldots A^k\mathbf{r}^0\right)$$

Krylov subspace methods try to find optimal approximation in this space

Well known methods:

- Conjugate Gradient (CG) for symmetric positive definite matrices
- GMRES
- others: BiCGSTAB, CGSquared, . . .

Condition number

Consider eigenvalue decomposition of A:

$$A\mathbf{v}_i = \lambda_i \mathbf{v}_i, \text{ with } i = 1, 2, \ldots, n$$

and decompose the error and residual using those eigenvectors:

$$\mathbf{e}^k = \sum_i \epsilon_i \mathbf{v}_i$$

$$\mathbf{r}^k = A\mathbf{e}^k = \sum_i \lambda_i \epsilon_i \mathbf{v}_i$$

the components of the error with large eigenvalue will be enlarged, and those with small eigenvalue diminished.

$$\text{Condition number: } \frac{\max_i |\lambda_i|}{\min_i |\lambda_i|}$$

Preconditioning

Combine stationary approach with approximate inverse matrix M^{-1} with Krylov methods

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha_k M^{-1}\mathbf{r}^k$$

this is equivalent with applying original Krylov method to solve

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$$

Thus now we should consider the condition number of $M^{-1}A$. The approximate inverse matrix, also called preconditioner, is useful if it brings down the condition number of $M^{-1}A$.

Preconditioned Krylov Subspace

Combines Krylov subspace methods:

- Conjugate Gradient (CG) for symmetric matrices
- GMRES
- others: BiCGSTAB, CGSq, ...

with suitable preconditioner:

- Jacobi
- Gauss Seidel
- Succesive Symmetric Over-Relaxation (SSOR)
- Incomplete LU (ILU)
- Multigrid methods
- many others

Error bounds

Error bounds are based on preconditioned residual: $M^{-1}r^k$

Absolute error tolerance:

$$\|M^{-1}r^k\| \leq \epsilon, \quad \texttt{-ksp\_atol}\ \epsilon$$

Relative error tolerance:

$$\|M^{-1}r^k\| \leq C\|M^{-1}b\|, \quad \texttt{-ksp\_rtol}\ C$$

Divergence tolerance:

$$\|M^{-1}r^k\| > D\|M^{-1}b\|, \quad \texttt{-ksp\_dtol}\ D$$

Solve the equation

- Choose iterative method
- Choose preconditioner
- Choose error tolerance

```
use solvers
use sparse_tools

! assemble A and rhs
...
! zero initial guess:
call zero(psi)
! set solver options
call set_solver_options(psi, ksptype='cg', pctype='sor', &
   rtol=1.0e-7, atol=0.0)
! solve the equation A \psi=rhs
call petsc_solve(psi, A, rhs)
```

Imperial College London

Trouble shooting

If a linear solve fails in fluidity it will:

- Put big warnings in the log.
- Tell you why it didn't succeed, e.g.: `KSP_DIVERGED_ITS`, `KSP_DIVERGED_DTOL`, `KSP_DIVERGED_NAN`.
- Dump the matrix equation it was trying to solve in a file called `matrixdump`.
- Stop the run at the end of the time step with the usual final vtk dump.

Trouble shooting

What to do if a linear solve fails:

- Check that your model is set up correctly, the problem is well-posed, right boundary conditions, etc.
- Check that your model results are reasonable before the first failing solve (for instance to see if it is not blowing up, or if there are NaNs).
- Check your mesh.
- Only if you are reasonably certain that is this the actual equation you want to solve try changing the solver options. For this purpose you can use `petsc_readnsolve` (see the wiki for instructions).

Closer look at the log

```
 Using PETSc to solve pressure.
 Inside petsc_solve_(block_)csr, solving for: Pressure
 Assembling matrix.
 Number of rows ==           8
 Number of blocks ==           1
 Matrix assembly completed.
 Using solver options defined at: /material_phase[0]/scalar_field::Pressure/prognostic/
 ksp_type:cg
 pc_type: sor
 ksp_max_it, ksp_atol, ksp_rtol, ksp_dtol:         1000   0.0000000000000000       9.99
 startfromzero: F
 Assembling RHS.
 RHS assembly completed.
 Assembling initial guess.
 Initial guess assembly completed.
 Entering solver.
 Out of solver.
Pressure PETSc reason of convergence: 2
Pressure PETSc n/o iterations: 5
 PETSc has solved pressure.
```