FEniCS Course

Lecture 19: FEniCS implementation

Contributors
Anders Logg

Key steps (linear PDEs)

1. Formulate linear variational problem: $a(u, v) = L(v)$
2. Assemble linear system: $A = A(a)$ and $b = b(L)$
3. Solve linear system: $U = A^{-1}b$

Key steps (nonlinear PDEs)

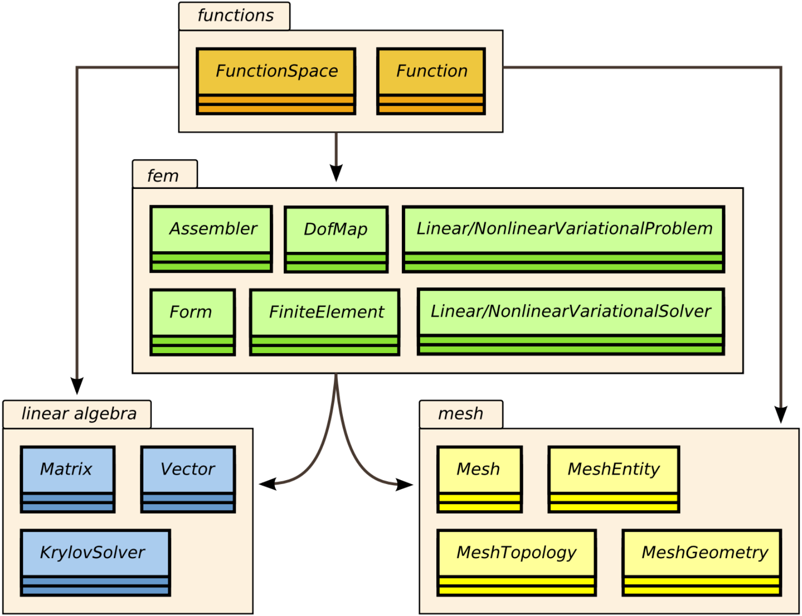
1. Formulate variational problem: $F(u) = 0$
2. Differentiate variational problem: $F' = \partial F / \partial u$
3. Solve nonlinear system:
    1. Assemble linear system: $A = A(F')$ and $b = b(F)$
    2. Solve linear system: $\delta U = -A^{-1} b$
    3. Update: $U \leftarrow U + \delta U$

Key steps for linear and nonlinear PDEs

1. Assemble linear system
2. Solve linear system

Key data structures

- Meshes: `Mesh`
- Sparse matrices and vectors:
  `Matrix`, `Vector`, `PETScMatrix`, `PETScVector`
- Functions: `Function`
- Dof maps: `DofMap`


functions
FunctionSpace
Function
fem
Assembler
DofMap
Linear/NonlinearVariationalProblem
Form
FiniteElement
Linear/NonlinearVariationalSolver
linear algebra
Matrix
Vector
KrylovSolver
mesh
Mesh
MeshEntity
MeshTopology
MeshGeometry

Key algorithms

- Assembling linear systems: `Assembler`
  - Mapping degrees of freedom: `DofMapBuilder`
  - Computing the element (stiffness) matrix: `ufc::tabulate_tensor`
  - Sparse matrix inseration: `Matrix.add()`
- Solving linear systems: `LinearSolver`, `PETScKrylovSolver`

Mesh data structure

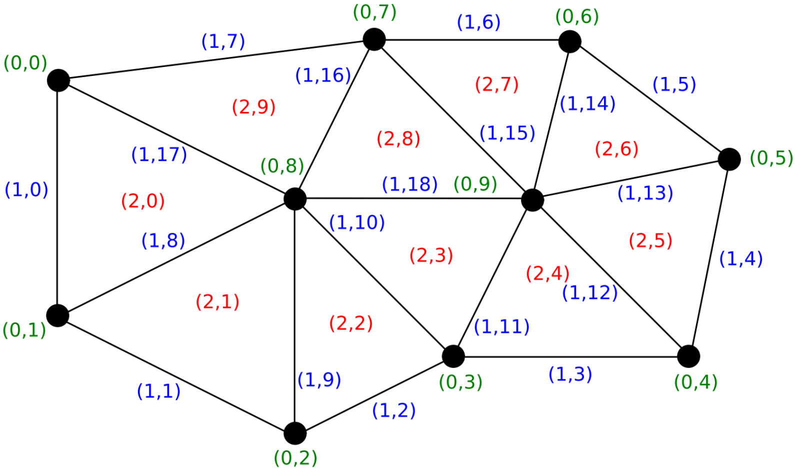
Separate mesh data into *topology* (connectivity) and *geometry* (coordinates).

From `dolfin/mesh/Mesh.h`:

C++ code

```
class Mesh
{
public:
  ...
private:
  MeshTopology _topology;
  MeshGeometry _geometry;
};
```

Mesh entities



Mesh topology

From `dolfin/mesh/MeshTopology.h`:

C++ code

```
class MeshTopology
{
public:
  ...
private:
    std::vector<std::vector<MeshConnectivity> >
        connectivity;
};
```

Mesh connectivity

From `dolfin/mesh/MeshConnectivity.h`:

C++ code

```
class MeshConnectivity
{
public:
  ...
private:
    std::vector<unsigned int> _connections;
    std::vector<unsigned int> _offsets;
};
```

Sparse matrix data structure

Sparse matrices in FEniCS are delegated to PETSc (or some other linear algebra backend).

Can otherwise be implemented using CRS (Compressed Row Storage):

C++ code

```
class Matrix
{
public:
  ...
private:
   double* data;
   unsigned int* cols;
   unsigned int* offsets;
};
```

Computing the sparse matrix A

- $a = a(u, v)$ is a bilinear form (form of arity 2)
- A is a sparse matrix (tensor of rank 2)

$$A_{ij} = a(\phi_j, \phi_i)$$

Note reverse order of indices!

Naive assembly algorithm

$A = 0$

for $i = 1, \ldots, N$

  for $j = 1, \ldots, N$

    $A_{ij} = a(\phi_j, \phi_i)$

  end for

end for

The element matrix

The global matrix A is defined by

$$A_{ij} = a(\phi_j, \phi_i)$$

The *element matrix* A_T is defined by

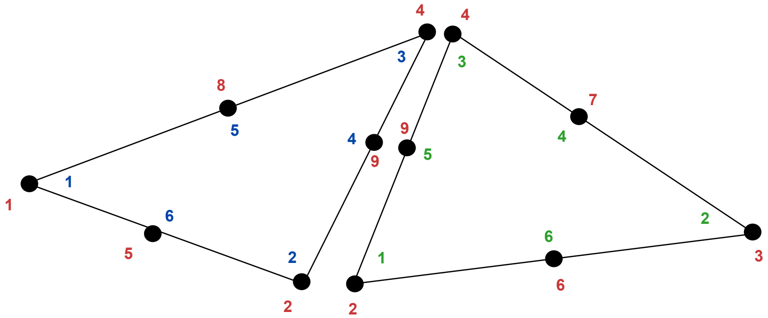
$$A_{T,ij} = a_T(\phi_j^T, \phi_i^T)$$

The local-to-global mapping

The global matrix ι_T is defined by

$$I = \iota_T(i)$$

where I is the *global index* corresponding to the *local index* i



The assembly algorithm

$A = 0$

for $T \in \mathcal{T}$

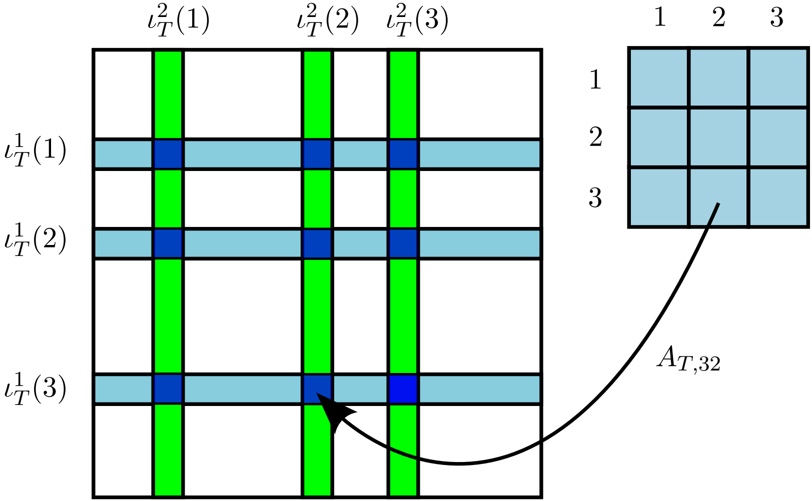
  Compute the element matrix A_T

  Compute the local-to-global mapping ι_T

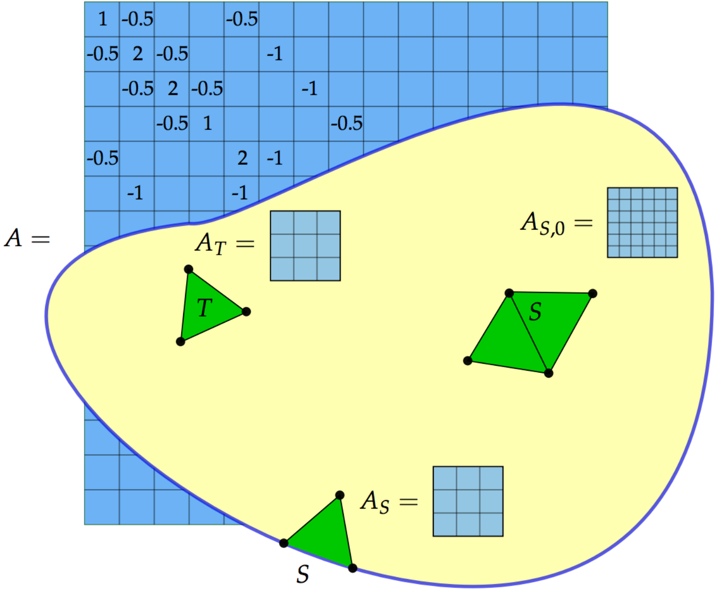
  Add A_T to A according to ι_T

end for

Adding the element matrix A_T

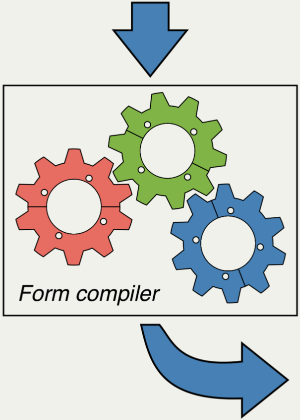


Cell integrals and facet integrals



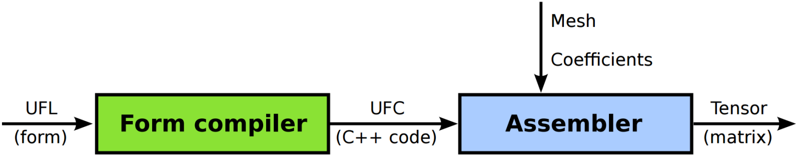
FFC generates code for A_T

`ffc -O3` "$-\Delta u = f$"

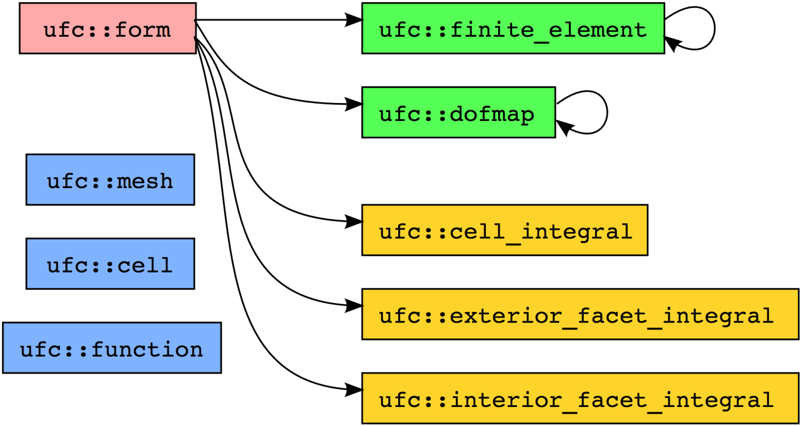



```
/// Tabulate the tensor for the contribution from a local cell
virtual void tabulate_tensor(double*  A,
                             const double * const *  w,
                             const double*  vertex_coordinates,
                             int cell_orientation) const
{
  // Number of operations (multiply-add pairs) for Jacobian data:      3
  // Number of operations (multiply-add pairs) for geometry tensor:    8
  // Number of operations (multiply-add pairs) for tensor contraction: 11
  // Total number of operations (multiply-add pairs):                  22

  // Compute Jacobian
  double J[4];
  compute_jacobian_triangle_2d(J, vertex_coordinates);

  // Compute Jacobian inverse and determinant
  double K[4];
  double detJ;
  compute_jacobian_inverse_triangle_2d(K, detJ, J);

  // Set scale factor
  const double det = std::abs(detJ);

  // Compute geometry tensor
  const double G0_0_0 = det*(K[0]*K[0] + K[1]*K[1]);
  const double G0_0_1 = det*(K[0]*K[2] + K[1]*K[3]);
  const double G0_1_0 = det*(K[2]*K[0] + K[3]*K[1]);
  const double G0_1_1 = det*(K[2]*K[2] + K[3]*K[3]);

  // Compute element tensor
  A[0] = 0.499999999999999*G0_0_0 + 0.5*G0_0_1 + 0.5*G0_1_0 + 0.5*G0_1_1;
  A[1] = -0.499999999999999*G0_0_0 - 0.5*G0_1_0;
  A[2] = -0.5*G0_0_1 - 0.5*G0_1_1;
  A[3] = -0.499999999999999*G0_0_0 - 0.5*G0_0_1;
  A[4] = 0.499999999999999*G0_0_0;
  A[5] = 0.5*G0_0_1;
  A[6] = -0.5*G0_1_0 - 0.5*G0_1_1;
  A[7] = 0.5*G0_1_0;
  A[8] = 0.5*G0_1_1;
}
```

Code generation chain



UFC data structures



The assembly implementation

From `dolfin/fem/Assembler.cpp`:

C++ code

```
void assemble(GenericTensor& A, const Form& a)
{
  ...
  for (CellIterator cell(mesh); !cell.end(); ++cell)
  {
    for (std::size_t i = 0; i < form_rank; ++i)
      dofs[i] = dofmaps[i]->cell_dofs(cell->index());

    integral->tabulate_tensor(ufc.A.data(), ...);

    A.add_local(ufc.A.data(), dofs);
  }
}
```

Iterative methods

Krylov subspace methods

- GMRES (Generalized Minimal RESidual method)
- CG (Conjugate Gradient method)
  - Works if A is symmetric and positive definite
- BiCGSTAB, MINRES, TFQMR, ...

Multigrid methods

- GMG (Geometric MultiGrid)
- AMG (Algebraic MultiGrid)

Preconditioners

- ILU, ICC, SOR, AMG, Jacobi, block-Jacobi, additive Schwarz, ...

Solving linear systems

Iterative linear solvers in FEniCS are delegated to PETSc (or some other linear algebra backend).

From `dolfin/la/PETScKrylovSolver.cpp`:

C++ code

```
std::size_t solve(PETScVector& x,
                  const PETScVector& b)
{
  ...
  // Solve system
  ierr =  KSPSolve(_ksp, b.vec(), x.vec());
  if (ierr != 0) petsc_error(ierr, __FILE__,
      "KSPSolve");
  ...
}
```

This function alone is 140 lines long. (!)