Introduction to Finite Element Methods

Andreas Dedner[1] Robert Klöfkorn[2]

(original slides by Peter Bastian with modifications from Claus-Justus Heine as part of the PDELab tutorial and Geilo Summerschool 2019, licensed under Creative Commons Attribution-ShareAlike 3.0 license)

[1] Institute of Mathematics, University of Warwick
[2] Center for Mathematical Sciences, Lund University

Motivation

- Start with an introduction to the finite element method (FEM) for solving Poisson's equation with piecewise linear "P_1" finite elements
- "Hello World!" for any numerical partial differential equation (PDE) solver framework!
- Gives necessary background for Python bindings of the `dune-fem` module

Challenges for PDE Software

- **Many different PDE applications**
  - Multi-physics
  - Multi-scale
  - Inverse modeling: parameter estimation, optimal control
  - Uncertainty quantification
- **Many different numerical solution methods**
  - No single method to solve all equations!
  - Different mesh types, mesh generation, mesh refinement
  - Higher-order approximations (polynomial degree)
  - Error control and adaptive mesh/degree refinement
  - Iterative solution of (non-)linear algebraic equations
- **High-performance Computing**
  - Single core performance: Often bandwidth limited
  - Parallelization through domain decomposition
  - Robustness w.r.t. to mesh size, model parameters, processors
  - Dynamic load balancing

$\Rightarrow$ **One software to do it all!**

Flexibility Requires Abstraction!

- DUNE-Fem and also DUNE-PDELab are based on an abstract formulation of the numerical scheme based on **residual forms**
- In order to implement a scheme it requires to put it to that form!
- Although you might be familiar with the FEM, you might not be familiar to the notation used here
- When you have mastered the abstraction you can solve complex problems with reasonable effort (see examples with UFL)
- Important feature: Orthogonality of concepts:
  - Dimension $d = 1, 2, 3, \ldots$
  - Linear and nonlinear
  - Stationary and Instationary
  - Scalar PDE and systems of PDEs
  - Uniform and adaptive mesh refinement of different types
  - Sequential and parallel

Introduction to the Finite Element Method

Strong Formulation of the PDE Problem

We solve Poisson's equation with inhomogeneous Dirichlet boundary conditions:

$$-\Delta u = f \qquad \text{in } \Omega, \tag{1a}$$

$$u = g \qquad \text{on } \partial\Omega, \tag{1b}$$

- $\Omega \subset \mathbb{R}^d$ is a polygonal domain in d-dimensional space
- A function $u \in C^2(\Omega) \cap C^0(\overline{\Omega})$ solving (1a), (1b) is called *strong solution*
- Inhomogeneous Dirichlet boundary conditions could be reduced to *homogeneous* ones: we will not do this!
- Proving existence and uniqueness of solutions of strong solutions requires quite restrictive conditions on f and g

Weak Formulation of the PDE Problem

Suppose u is a strong solution and take *any test function* $v \in C^1(\Omega) \cap C^0(\overline{\Omega})$ with $v = 0$ on $\partial\Omega$ then:

$$\int_\Omega (-\Delta u) v \, dx = \underbrace{\int_\Omega \nabla u \cdot \nabla v \, dx}_{=: a(u,v)} = \underbrace{\int_\Omega f v \, dx}_{=: l(v)} .$$

Question: Is there a vector space of functions V with $V_g = \{v \in V : v = g \text{ on } \partial\Omega\}$ and $V_0 = \{v \in V : v = 0 \text{ on } \partial\Omega\}$ such that the problem

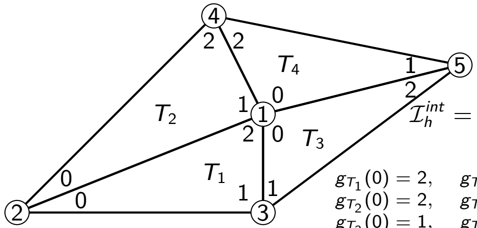
$$u \in V_g : \qquad a(u,v) = l(v) \qquad \forall v \in V_0 \tag{2}$$

has a unique solution?
Answer: Yes, $V = H^1(\Omega)$. This u is called *weak solution*.
Advantage: Weak solutions do exist under less restrictive conditions on the data.

The Finite Element Method

- The finite element method (FEM) is one method for the numerical solution of PDEs
- Others are the finite volume method (FVM) or the finite difference method (FDM)
- The FEM is based on the weak formulation derived above
- Its basic idea is to replace the space V by a *finite-dimensional space* V_h!
- The construction of these finite-dimensional spaces needs some preparations . . .

Finite Element Mesh



$\mathcal{I}_h^{int} = \{1\}, \mathcal{I}_h^{\partial\Omega} = \{2,3,4,5\}$

$g_{T_1}(0) = 2,$	$g_{T_1}(1) = 3,$	$g_{T_1}(2) = 1,$
$g_{T_2}(0) = 2,$	$g_{T_2}(1) = 1,$	$g_{T_2}(2) = 4,$
$g_{T_3}(0) = 1,$	$g_{T_3}(1) = 3,$	$g_{T_3}(2) = 5,$
$g_{T_4}(0) = 1,$	$g_{T_4}(1) = 5,$	$g_{T_4}(2) = 4.$

- A mesh consists of ordered sets of vertices and elements:

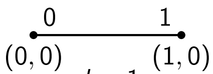
$$\mathcal{X}_h = \{x_1, \ldots, x_N\} \subset \mathbb{R}^d, \quad \mathcal{T}_h = \{T_1, \ldots, T_M\}$$

- *Simplicial element*: $T = \text{convex\_hull}(x_{T,0}, \ldots, x_{T,d})$
- *Conforming*: Intersection is subentity
- *Local to global map* : $\mathcal{M}_T : \{0, \ldots, d\} \to \mathcal{N}$

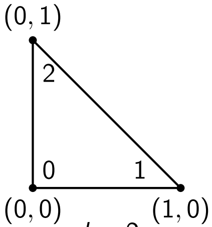
$$\forall T \in \mathcal{T}_h, 0 \le i \le d : \mathcal{M}_T(i) = j \Leftrightarrow x_{T,i} = x_j.$$

- *Interior and boundary vertex index sets*: $\mathcal{I}_h = \mathcal{I}_h^{int} \cup \mathcal{I}_h^{\partial\Omega}$, $\mathcal{I}_h^{int} = \{i \in \mathcal{I}_h \,:\, x_i \in \Omega\}, \mathcal{I}_h^{\partial\Omega} = \{i \in \mathcal{I}_h \,:\, x_i \in \partial\Omega\}$

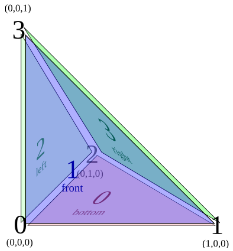
Reference Element and Element Transformation



$d = 1$



$d = 2$



$d = 3$

- $\hat{T}^d$ is the reference simplex in d space dimensions
- The mesh $\mathcal{T}_h$ is called *affine* if for every $T \in \mathcal{T}_h$ there is an affine linear map $F_T : \hat{T} \to T$,

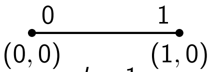
$$F_T(\hat{x}) = B_T\hat{x} + q_T$$

  with

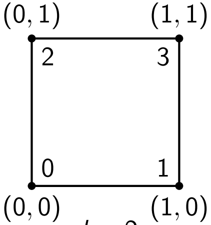
$$\forall i \in \{0, \ldots, d\} : F_T(\hat{x}_i) = x_{T,i}$$

Reference Element and Element Transformation

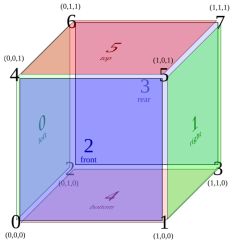
cont. Cube Mesh



$d = 1$



$d = 2$



$d = 3$

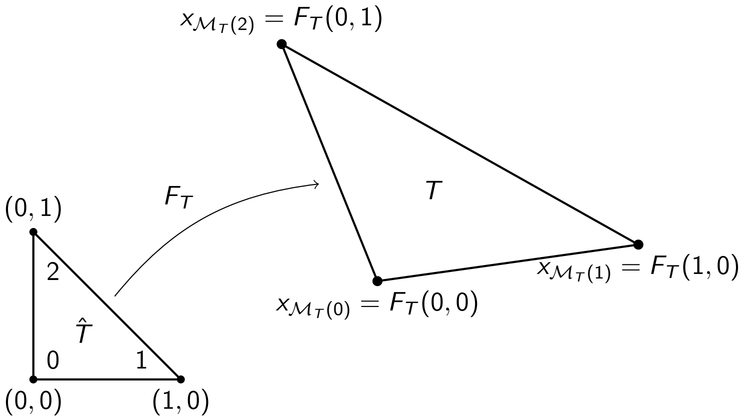
- $\hat{Q}^d$ is the reference cube in d space dimensions
- normally one uses d-linear interpolation F_Q between the vertices of $\hat{Q}^d$ and a d-"cuboid",

$$F_Q(\hat{x}_i) = x_{g_Q(i)}, \quad i = 0, ..., (2^d - 1)$$

- the mapping between $\hat{Q}^d$ and a general d-"cuboid" Q cannot be linear in general

Reference Element and Element Transformation

cont.


$x_{\mathcal{M}_T(2)} = F_T(0,1)$
T
$x_{\mathcal{M}_T(1)} = F_T(1,0)$
$x_{\mathcal{M}_T(0)} = F_T(0,0)$
F_T
$(0,1)$
2
$\hat{T}$
0
1
$(0,0)$
$(1,0)$

- Reference mapping $F_T : \hat{T} \to T : \hat{x} \mapsto F_T(\hat{x}) = B_T\hat{x} + x_{\mathcal{M}_T(0)}$.
- Integration element $|\det(B_T)| = |\det(\hat{\nabla} F_T)|$.

Piecewise Linear Finite Element Space

- The idea of the *conforming* FEM is to solve the weak problem in *finite-dimensional* function spaces $V_h \subset V$:

$$u_h \in V_{h,g} : \quad a(u_h, v) = l(v) \quad \forall v \in V_{h,0}.$$

- A particular choice is the space of *piecewise linear* functions

$$V_h(\mathcal{T}_h) = \{v \in C^0(\overline{\Omega}) : \forall T \in \mathcal{T}_h : v|_T \in \mathbb{P}_1^d\}$$

  where $\mathbb{P}_1^d = \{p : \mathbb{R}^d \to \mathbb{R} : p(x) = a^T x + b, a \in \mathbb{R}^d, b \in \mathbb{R}\}$
- One can show $\dim V_h = N = \dim \mathcal{X}_h$ and $V_h \subset H^1(\Omega)$
- *Lagrange* basis functions:

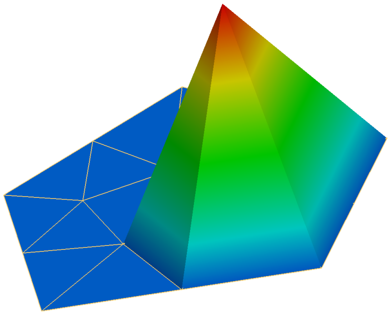
$$\Phi_h = \{\phi_1, \dots, \phi_N\}, \quad \forall i, j \in \mathcal{I}_h : \phi_i(x_j) = \delta_{i,j}$$

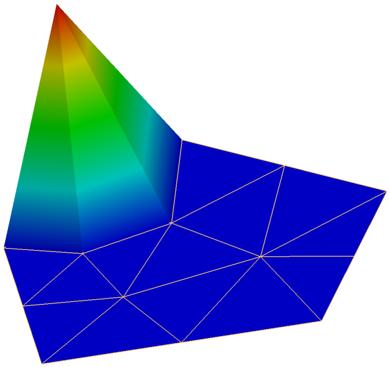
- *Test and Ansatz (Trial) spaces*:

$$V_{h,0} = \{v \in V_h : \forall i \in \mathcal{I}_h^{\partial\Omega} : v(x_i) = 0\},$$
$$V_{h,g} = \{v \in V_h : \forall i \in \mathcal{I}_h^{\partial\Omega} : v(x_i) = g(x_i)\} = v_{h,g} + V_{h,0}$$

Examples of Finite Element Functions

Here in two space dimensions:





Due to their shape they are often called *hat functions*

Construction of Finite Element Basis Functions

- Define “shape functions” $\hat{\phi}_i$ on the reference element
- Construct global basis functions ϕ_k for $k = \mathcal{M}_T(i)$ by a “pull-back”

$$\phi_k|_T(x) := \hat{\phi}_i(F_T^{-1}(x)) \Leftrightarrow \phi_k|_T(F_T(\hat{x})) = \hat{\phi}_i(\hat{x}).$$

Example for Lagrange functions on simplices: use e.g. “barycentric coordinates”

$$\lambda_0(\hat{x}) = 1 - \sum_{i=1}^{d} \hat{x}_i, \quad \lambda_i(\hat{x}) = \hat{x}_i \ (i > 0), \quad F_T(\hat{x}) = \sum_{i=0}^{d} \lambda_i(\hat{x})\, x_{\mathcal{M}_T(i)}.$$

- Linear: $\hat{\phi}_i = \lambda_i$, $i = 0, \ldots, d$
- Quadratic: ...

Finite Element Solution

Inserting a *basis representation* $u_h = \sum_{j=1}^{N} (z)_j \phi_j$ results in

$$\begin{aligned} a(u_h, v) &= l(v) \quad \forall v \in V_{h,0} \quad \text{(discrete weak problem)},\\ \Leftrightarrow a\left(\sum_{j=1}^{N} (z)_j \phi_j, \phi_i\right) &= l(\phi_i) \quad \forall i \in \mathcal{I}_h^{int} \quad \text{(insert basis, linearity)},\\ \Leftrightarrow \sum_{j=1}^{N} (z)_j a(\phi_j, \phi_i) &= l(\phi_i) \quad \forall i \in \mathcal{I}_h^{int} \quad \text{(linearity)}. \end{aligned}$$

Together with the condition $u_h \in V_{h,g}$ expressed as

$$u_h(x_i) = z_i = g(x_i) \quad \forall i \in \mathcal{I}_h^{\partial\Omega}$$

this forms a system of linear equations

$$Ax = b$$

where

$$(A)_{i,j} = \begin{cases} a(\phi_j, \phi_i) & i \in \mathcal{I}_h^{int} \\ \delta_{i,j} & i \in \mathcal{I}_h^{\partial\Omega} \end{cases}, \quad (b)_i = \begin{cases} l(\phi_i) & i \in \mathcal{I}_h^{int} \\ g(x_i) & i \in \mathcal{I}_h^{\partial\Omega} \end{cases}.$$

Solution of Linear Systems

- *Exact* solvers based on Gaussian elimination
- This may become inefficent for *sparse* linear systems
- *Iterative* methods (hopefully) produce a convergent sequence

$$\lim_{k\to\infty} z^k = z$$

- A very simple example is *Richardson's* iteration:

$$z^{k+1} = z^k + \omega(b - Az^k)$$

  requiring only *matrix-vector products*
- Another well known class of iterative solvers are Krylov methods requiring also only matrix-vector products

Three Steps to Solve the FE Problem

1. Assembling the matrix A. This mainly involves the computation of the matrix elements $a(\phi_j, \phi_i)$ and storing them in an appropriate data structure.
2. Assembling the right hand side vector b. This mainly involves evaluations of the right hand side functional $l(\phi_i)$.
3. *Alternatively:* Perform a matrix free operator evaluation $y = Az$. This involves evaluations of $a(u_h, \phi_i)$ for all test functions ϕ_i and a given function u_h due to:

$$
\begin{aligned}
(Az)_i &= \sum_{j=1}^{N} (A)_{i,j}(z)_j = \sum_{j=1}^{N} a(\phi_j, \phi_i)(z)_j \\
&= a\left(\sum_{j=1}^{N} (z)_j \phi_j, \phi_i\right) = a(u_h, \phi_i)
\end{aligned}
$$

We now discuss *how* these steps may be implemented.

Four Important Tools

1. Transformation formula for integrals. For $T \in \mathcal{T}_h$:

$$\int_T y(x)\,dx = \int_{\hat{T}} y(F_T(\hat{x}))|\det B_T|\,dx.$$

2. Midpoint rule on the reference element:

$$\int_{\hat{T}} q(\hat{x})\,dx \approx q(\hat{S}_d)w_d$$

   (More accurate formulas are used later)

3. Basis functions via shape function transformation:

$$\hat{\phi}_0(\hat{x}) = 1 - \sum_{i=1}^{d}(\hat{x})_i, \quad \hat{\phi}_i(\hat{x}) = (\hat{x})_i, i > 0, \quad \phi_{T,i}(F_T(\hat{x})) = \hat{\phi}_i(\hat{x})$$

4. Computation of gradients. For any $w(F_T(\hat{x})) = \hat{w}(\hat{x})$:

$$B_T^T \nabla w(F_T(\hat{x})) = \hat{\nabla}\hat{w}(\hat{x}) \quad \Leftrightarrow \quad \nabla w(F_T(\hat{x})) = B_T^{-T}\hat{\nabla}\hat{w}(\hat{x}).$$

Assembly of Right Hand Side I

In computing $(b)_i$ only the following elements are involved:

$$C(i) = \{(T, m) \in \mathcal{T}_h \times \{0, \dots, d\} \, : \, \mathcal{M}_T(m) = i\}$$

Then

$$\begin{aligned}
(b)_i &= l(\phi_i) = \int_\Omega f \phi_i \, dx && \text{(definition)} \\
&= \sum_{T \in \mathcal{T}_h} \int_T f \phi_i \, dx && \text{(use mesh)} \\
&= \sum_{(T,m) \in C(i)} \int_{\hat{T}} f(F_T(\hat{x})) \hat{\phi}_m(\hat{x}) |\det B_T| \, dx && \text{(localize)} \\
&= \sum_{(T,m) \in C(i)} f(F_T(\hat{S}_d)) \hat{\phi}_m(\hat{S}_d) |\det B_T| w_d \; + \text{err.} && \text{(quadrature)}
\end{aligned}$$

Assembly of Right Hand Side II

- Now we need to perform these computations *for all* $i \in \mathcal{I}_h^{int}$!
- Collect *element-local* computations:

$$(b_T)_m = f(F_T(\hat{S}_d))\hat{\phi}_m(\hat{S}_d)|\det B_T| w_d \quad \forall m = 0, \dots, d$$

- Define restriction matrix $R_T : \mathbb{R}^N \to \mathbb{R}^{d+1}$ with

$$(R_T x)_m = (x)_i \quad \forall\, 0 \le m \le d,\, \mathcal{M}_T(m) = i,$$

- Then

$$b = \sum_{T \in \mathcal{T}_h} R_T^T b_T.$$

Assembly of Global Stiffness Matrix I

In computing $(A)_{i,j}$ only the following elements are involved:

$$C(i,j) = \{(T,m,n) \in \mathcal{T}_h \times \{0,\dots,d\} \,:\, \mathcal{M}_T(m) = i \wedge \mathcal{M}_T(n) = j\}$$

Then

$$\begin{aligned}
(A)_{i,j} &= a(\phi_j,\phi_i) = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, dx && \text{(definition)}\\
&= \sum_{T\in\mathcal{T}_h} \int_T \nabla\phi_j \cdot \nabla\phi_i \, dx && \text{(use mesh)}\\
&= \sum_{(T,m,n)\in C(i,j)} \int_{\hat{T}} (B_T^{-T}\hat{\nabla}\hat{\phi}_n(\hat{x})) \cdot (B_T^{-T}\hat{\nabla}\hat{\phi}_m(\hat{x})) |\det B_T| \, d\hat{x} && \text{(localize)}\\
&= \sum_{(T,m,n)\in C(i,j)} (B_T^{-T}\hat{\nabla}\hat{\phi}_n(\hat{S}_d)) \cdot (B_T^{-T}\hat{\nabla}\hat{\phi}_m(\hat{S}_d)) |\det B_T| w_d. && \text{(quadrature)}
\end{aligned}$$

Assembly of Global Stiffness Matrix II

- Now we need to perform these computations for *all* matrix entries!
- Define the $d \times d+1$ matrix of shape function gradients

$$\hat{G} = \left[\hat{\nabla}\hat{\phi}_0(\hat{S}_d)), \dots, \hat{\nabla}\hat{\phi}_d(\hat{S}_d))\right].$$

  and the matrix of transformed gradients

$$G = B_T^{-T}\hat{G}$$

- Define the *local stiffness matrix*

$$A_T = G^T G |\det B_T| w_d.$$

- Then

$$A = \sum_{T \in \mathcal{T}_h} R_T^T A_T R_T.$$

Implementation Summary

- All necessary steps in the solution procedure have the following general form:

```
1: for $T \in \mathcal{T}_h$ do                        ▷ loop over mesh elements
2:     $z_T = R_T z$                                   ▷ load element data
3:     $q_T = \text{compute}(T, z_T)$                  ▷ element local computations
4:     Accumulate$(q_T)$                   ▷ store result in global data structure
5: end for
```

- DUNE-Fem provides a generic *assembler (galerkin scheme)* that performs all these steps, except (3) which needs to be supplied by the implementor using UFL forms
- All these concepts carry over to
  - Nonlinear problems
  - Time-dependent problems
  - Systems of PDEs
  - High-order methods
  - Other schemes such as FVM, nonconforming FEM
  - Parallel computations

Sparse Numerical Linear Algebra

Problem: Number of matrix entries scale with N^2 where N is the number of unknowns.

dense matrix

$$\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

Scales with N^2.

Sparse Numerical Linear Algebra

Problem: Number of matrix entries scale with N^2 where N is the number of unknowns.

dense matrix

$$\begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}$$

Scales with N^2.

sparse matrix

$$\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & 0 & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & 0 & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{pmatrix}$$

Scales with $3N$.

Density and Sparsity

In numerical analysis and scientific computing, a **sparse** matrix is a matrix in which most of the elements are zero.
By contrast, if most of the elements are nonzero, then the matrix is considered **dense**.

Definition 1
Let $A \in \mathbb{R}^{n \times m}$ and
$A_{nz} := \{(i,j) \mid a_{ij} \neq 0 \text{ for } i = 1, \ldots, n \text{ and } j = 1, \ldots, m\}$.
We call

$$D := \frac{|A_{nz}|}{n\, m}$$

the density of A.
If $D < \frac{1}{2}$ we call A sparse.

Sparse Matrix Formats

- Dictionary of keys (DOK) where (row, column)-pairs are mapped to a value
- List of lists (LIL) where one list per row is stored, with each entry containing the column index and the value.
- Coordinate list (COO) stores a list of (row, column, value) tuples.
- Compressed sparse row (CSR, CRS or Yale format) where three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices are stored. It is similar to COO, but compresses the row indices, hence the name. This format is very efficient for matrix-vector multiplication.
- Compressed sparse column (CSC or CCS) is similar to CSR but compresses the column indices.
- More formats for special structures like **banded**, **diagonal**, etc.

Sparse Matrix Formats

- Dictionary of keys (DOK) where (row, column)-pairs are mapped to a value
- List of lists (LIL) where one list per row is stored, with each entry containing the column index and the value.
- Coordinate list (COO) stores a list of (row, column, value) tuples.
- Compressed sparse row (CSR, CRS or Yale format) where three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices are stored. It is similar to COO, but compresses the row indices, hence the name. This format is very efficient for matrix-vector multiplication.
- Compressed sparse column (CSC or CCS) is similar to CSR but compresses the column indices.
- More formats for special structures like **banded**, **diagonal**, etc.

Sparse Matrix Formats

- Dictionary of keys (DOK) where (row, column)-pairs are mapped to a value
- List of lists (LIL) where one list per row is stored, with each entry containing the column index and the value.
- Coordinate list (COO) stores a list of (row, column, value) tuples.
- Compressed sparse row (CSR, CRS or Yale format) where three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices are stored. It is similar to COO, but compresses the row indices, hence the name. This format is very efficient for matrix-vector multiplication.
- Compressed sparse column (CSC or CCS) is similar to CSR but compresses the column indices.
- More formats for special structures like **banded**, **diagonal**, etc.

Sparse Matrix Formats

- Dictionary of keys (DOK) where (row, column)-pairs are mapped to a value
- List of lists (LIL) where one list per row is stored, with each entry containing the column index and the value.
- Coordinate list (COO) stores a list of (row, column, value) tuples.
- Compressed sparse row (CSR, CRS or Yale format) where three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices are stored. It is similar to COO, but compresses the row indices, hence the name. This format is very efficient for matrix-vector multiplication.
- Compressed sparse column (CSC or CCS) is similar to CSR but compresses the column indices.
- More formats for special structures like **banded**, **diagonal**, etc.

Sparse Matrix Formats

- Dictionary of keys (DOK) where (row, column)-pairs are mapped to a value
- List of lists (LIL) where one list per row is stored, with each entry containing the column index and the value.
- Coordinate list (COO) stores a list of (row, column, value) tuples.
- Compressed sparse row (CSR, CRS or Yale format) where three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices are stored. It is similar to COO, but compresses the row indices, hence the name. This format is very efficient for matrix-vector multiplication.
- Compressed sparse column (CSC or CCS) is similar to CSR but compresses the column indices.
- More formats for special structures like **banded**, **diagonal**, etc.

Sparse Matrix Formats

- Dictionary of keys (DOK) where (row, column)-pairs are mapped to a value
- List of lists (LIL) where one list per row is stored, with each entry containing the column index and the value.
- Coordinate list (COO) stores a list of (row, column, value) tuples.
- Compressed sparse row (CSR, CRS or Yale format) where three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices are stored. It is similar to COO, but compresses the row indices, hence the name. This format is very efficient for matrix-vector multiplication.
- Compressed sparse column (CSC or CCS) is similar to CSR but compresses the column indices.
- More formats for special structures like **banded**, **diagonal**, etc.

Sparse Matrix Formats

- Dictionary of keys (DOK) where (row, column)-pairs are mapped to a value
- List of lists (LIL) where one list per row is stored, with each entry containing the column index and the value.
- Coordinate list (COO) stores a list of (row, column, value) tuples.
- Compressed sparse row (CSR, CRS or Yale format) where three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices are stored. It is similar to COO, but compresses the row indices, hence the name. This format is very efficient for matrix-vector multiplication.
- Compressed sparse column (CSC or CCS) is similar to CSR but compresses the column indices.
- More formats for special structures like **banded**, **diagonal**, etc.

Scipy.sparse: Create CSR from COO format

Python code

```python
from scipy.sparse import csr_matrix
import numpy as np

# row indices
rowind = np.array([0, 0, 1, 1, 1, 2, 2, 2, 3, 3])

# column indices
colind = np.array([0, 1, 0, 1, 2, 1, 2, 3, 2, 3])

# matrix values
values = np.array([-2.,1.,1.,-2.,1.,1.,-2.,1.,1.,-2.])

# create sparse from COO format
# all arrays have the same length
A = csr_matrix( (values, (rowind, colind) ) )

# let's see what we got
print (A)
```

Further Reading

Tutorial dune-fem
https://dune-project.org/sphinx/content/sphinx/dune-fem/
Accessed April 4, 2022.