Efficient C++ implementation of custom FEM kernel with Eigen

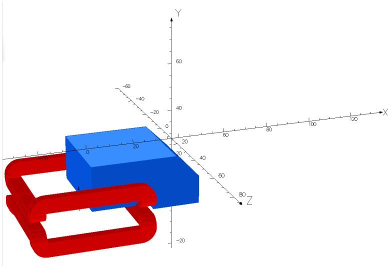
Mikhail Sizov

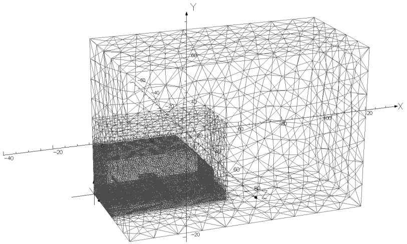
INP SB RAS

January 31, 2019

Outline

Design of electromagnets





Problem statement

Lens quality is determined by features of magnetic field (e.g. uniformity)

- Qualified labor, but includes a lot of trial and error
- Simulating fields in 3D with existing tools takes **up to several days**
- Existing software can't reuse intermediate calculations

Goals:

- Reduce human labor
- Reduce total count of full 3D modeling

Solution: randomizing geometries with genetic algorithms
Takes thousands of iterations to get good results, need to be *fast*

Problem statement

Use custom 2D finite element method kernel:

- Calculates field in 2D projection of magnet
- Single evaluation takes $O(n^2)$ instead of $O(n^3)$
- Caches calculations
- Uses uniformity of magnet field in target area as a quality measure

Cons:

- Less precision
- Suitable for limited set of geometries (either long bodies or solids of revolution)

Problem statement in 2D

Coil current J_z generates magnetic field (B)

Ferromagnetic core with permeability μ forms magnetic field

For 2D projection of long magnet, A_z (magnetic potential along Z) fully determines magnetic field
Known:

- Maxwell equations
- Object geometry and properties (e.g. permeability μ or BH curve of material)
- Current J_z
- Boundary conditions for A_z (e.g. potential is zero at ∞)

Unknown: Magnetic potential A_z in rest of domain area

Finite element method

Solves partial differential problems in *finite* domain, e.g.

- Fluid flow
- Heat transfer
- Structural analysis

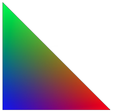
Finite element method

Why FEM?

- Suited well for complex geometries, works with irregular or adaptive meshes (density can be proportional to spatial derivative of target value, more value changes in area would lead to more dense mesh)
- Other method is numerical integration, usually has regular mesh (equal distance between discrete points)

Finite element method

- Numerical computations (approximate)
- Differential equations translated to **algebraic equation systems**
- Domain geometry is approximated with finite elements (e.g. triangles)
- Method works when differential equations on linear functions result in polynomial equations.



Target variable (A_z) is linearly interpolated within element, so that A_z is continuous function

Eigen

Eigen is a C++ *header-only* template library for linear algebra.
Operates with *vectors* and *matrices*
Other popular linear algebra libraries are:

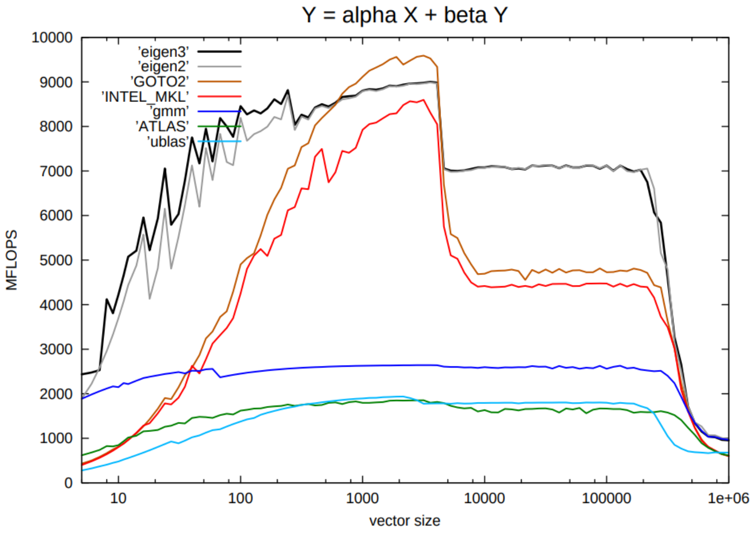
- OpenBLAS (C++)
- Armadilo (C++)
- scipy (python)
- cuBLAS (CUDA)

Why Eigen?

- Fixed and dynamic-sized matrices and vectors
- Sparse/dense matrices and vectors
- Multi-platform, multi-compiler
- Expression templates (remove temporaries, lazy evaluation)
- Optimized fixed-size matrices: (no dynamic memory allocation, unrolled loops)
- Standard numeric types (std::complex, integers, float)
- Interface for custom types

Why Eigen?

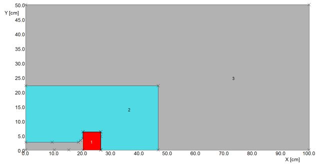
Benchmark provided on Eigen website (higher is better):

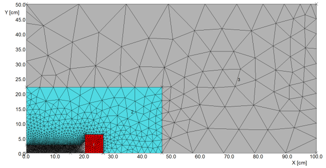


FEM first step : create a Mesh

Start with geometry approximation by simple shapes (in 2D: triangles, convex quadrilaterals)

Geometry approximation is called *mesh*

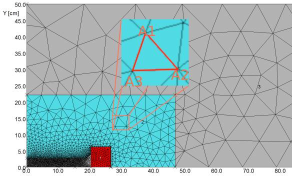




FEM first step : create a Mesh

Each single triangle is a *finite element*, it has:

- 3 triangle vertices (nodes)
- Vertix coordinates (x_1, y_1),(x_2, y_2),(x_3, y_3)
- Constant material properties within: J_z or μ
- Node values $(A^{el}_{z1}$,A^{el}_{z2},$A^{el}_{z3})$



Target A^{el}_z is linearly interpolated between nodes, e.g.
$A^{el}_z(x, y) = a + bx + cy$, more on that later
Node values either known from boundary conditions or unknowns
Node values shared between elements (A_z is continuous function)

Mesh

```
 Eigen::Vector2d p1 = {x1, y1}; // point coordinates
 Eigen::Vector2d p2 = {x2, y2};
 Eigen::Vector2d p3 = {x3, y3};
 // Vector2d - static size of 2 (up to 4 defined)
 // "d" stands for double (could also be int, float, double or
      std::complex<T>)

 // VectorXd - uses dynamic memory allocation
typedef Matrix <double,2,1> Eigen::Vector2d;
// Implemented as dense Matrix with size known at compile time
Eigen::Matrix<double, 2, 3> geometry;
geometry << x1,y1,x2,y2,x3,y3; // Dense matrix, other way
```

Element's geometry can also be stored in dense matrix:

$$\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix}$$

Shape functions

Interpolation within element is linear. By definition it must return node values on node coordinates $A_z^{el}(x_i, y_i) = A_{zi}^{el}$
Let's introduce set of 3 interpolating functions $\Phi_i(x, y)$ (shape function), that is:

- Linear $\Phi_i(x, y) = \alpha_i + \beta_i x + \gamma_i y$
- Equals 1 on it's own node $\Phi_i(x_i, y_i) = 1$
- Equals 0 on other 2 nodes $\Phi_i(x_j, y_j) = 0$, given $i \neq j$

Now, we can express $A_z^{el}(x, y)$ as :

$$A_z^{el}(x, y) = \begin{bmatrix} \Phi_1(x, y) & \Phi_2(x, y) & \Phi_3(x, y) \end{bmatrix} \begin{bmatrix} A_{z1} \\ A_{z2} \\ A_{z3} \end{bmatrix} = \sum_{i=1}^{3} \Phi_i(x, y) A_{zi}^{el}$$

Shape functions(implementation)

Shape functions coefficient matrix $\Phi_i(x, y) = \alpha_i + \beta_i x + \gamma_i y$ is known from geometry:

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \\ \gamma_1 & \gamma_2 & \gamma_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1}$$

Shape functions(implementation)

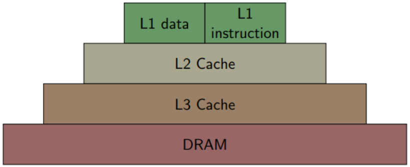
Element's shape functions coefficients can be calculated from element coordinates:

```
Eigen::Matrix<double, 3, 3, RowMajor> geometry;
// Dense 3x3 matrix to store geometry
geometry.block<3,1>(0,0) = Vector3d::Ones(); //fill first
    column
geometry.block<1,2>(0,1) = p1; // block assignments
geometry.block<1,2>(1,1) = p2;
geometry.block<1,2>(2,1) = p3;
auto shape = geometry.inverse();
// get actual shape coefficients
```

Specific memory layout can improve performance (ColMajor vs RowMajor):

$$mCol = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad mRow = \begin{bmatrix} \boxed{1 \quad 2 \quad 3} \\ 4 \quad 5 \quad 6 \\ 7 \quad 8 \quad 9 \end{bmatrix}$$

Typical structure of CPU memory caches



- L1 cache reference 1ns
- L2 cache reference 4ns
- L3 cache reference 15ns
- Main memory reference 100 ns

Modern CPU have SIMD vectorization instructions e.g. AVX set
AVX-256 operates with 256bit of data (4 double values), data is loaded from continuous memory region

Shape functions(implementation)

10x10 matrix of doubles, A * B = C

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

X

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

C[5][6] = 41*6+42*16+43*26...

Cache line has 64B, data is prefetched

4B	4B	4B	4B	4B	4B	4B	4B	4B	4B	4B	4B	4B	4B	4B	4B

Shape functions(implementation)

Best option, no extra cache misses:

RowMajor	🟩 41	🟩 42	🟩 42	🟩 43	🟩 43	🟩 44	🟩 44	🟩 45
X								
ColMajor	🟩 6	🟩 16	🟩 26	🟩 36	🟩 46	🟩 56	🟩 66	🟩 76

Common variant, matrix type is the same, second matrix has cache misses

RowMajor	🟩 41	🟩 42	🟩 42	🟩 43	🟩 43	🟩 44	🟩 44	🟩 45
X								
RowMajor	🟩 6	🟥 7	🟥 8	🟥 9	🟥 10	🟥 11	🟥 12	🟥 13

Worst scenario, every hit is cache miss

ColMajor	🟩 41	🟥 51	🟥 61	🟥 71	🟥 81	🟥 91	🟥 2	🟥 12
X								
RowMajor	🟩 6	🟥 7	🟥 8	🟥 9	🟥 10	🟥 11	🟥 12	🟥 13

System total energy

Shape functions + potential values at nodes define A_z in domain (solves problem)

How to calculate unknown A_z at nodes?

System total energy

Let's write potential energy per unit length of magnet field for an element:

$$U(A_z^{el}(x,y)) = \frac{1}{4\pi\mu}\int \left|\nabla A_z^{el}(x,y)\right|^2 dS + \frac{1}{c}\int A_z^{el}(x,y)\cdot J_z dS$$

Substituting shape functions Φ_i and vector of node values $\vec{A}_z^{el}$ into $A_z^{el}(x,y)$, we would get:

$$U(A_z) = \frac{1}{4\pi\mu}\vec{A}_z^{elT} C \vec{A}_z^{el} + \frac{S}{3c} J_z \sum_{i=1}^{3} A_{zi}^{el}$$

μ – element permeability
S – element square
C – element coupling matrix (from shape functions)

$$C = \begin{bmatrix} \beta_1 & \gamma_1 \\ \beta_2 & \gamma_2 \\ \beta_3 & \gamma_3 \end{bmatrix} \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \\ \gamma_1 & \gamma_2 & \gamma_3 \end{bmatrix}$$

System total energy

Total system energy can be obtained from sum of each element's energies:

$$U(A_z) = M\vec{A}_z^{elT} C \vec{A}_z^{el} + J\vec{A}_z^{el}$$

Looks similar, differences:
M – vector storing values proportional to node permeability constants
J – vector storing values proportional to node current (can have zeros inside if no current within element)
Matrix C stores global coefficient matrix

Global coefficient matrix

Global coefficient matrix is sparse (real example for 9x9 regular mesh):

$$C = \begin{bmatrix} -2 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & -2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & -4 & 2 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 2 & -8 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 2 & -8 & 2 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 2 & -8 \end{bmatrix}$$

1. Most values are zeros
2. Symmetrical
3. Typical node count for 2D is 250k, about 1k of *non-zeros*

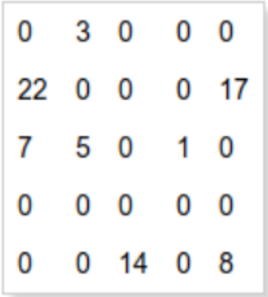
Global coefficient matrix

Dedicated types exist for efficiently storing sparse matrices:

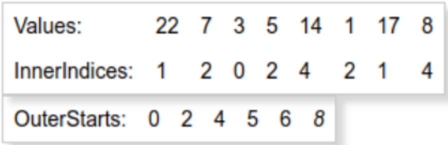
```
typedef Eigen::Triplet<double> T;
std::vector<T> tripletList;
tripletList.reserve(estimation_of_entries);
for(...)
{
 // ...
 tripletList.push_back(T(i,j,v_ij));
}
SparseMatrixType mat(rows,cols);
mat.setFromTriplets(tripletList.begin(), tripletList.end());
// mat is ready to go!
```

Global coefficient matrix

Sparse matrix memory layout:



0	3	0	0	0
22	0	0	0	17
7	5	0	1	0
0	0	0	0	0
0	0	14	0	8



Values:	22	7	3	5	14	1	17	8
InnerIndices:	1	2	0	2	4	2	1	4

OuterStarts:	0	2	4	5	6	8

- Values: stores the coefficient values of the non-zeros.
- InnerIndices: stores the row (in row major specialization) indices of the non-zeros.
- OuterStarts: stores for each column (resp. row) the index of the first non-zero in the previous two arrays.

Obtaining solution

Let's use physics principle of minimal potential energy:
From all possible values of free A_{zi}^{el} parameters, we need to choose set that gives minimal energy, so derivative of energy by each of A_{zi}^{el} is zero:

$$\frac{\partial U}{\partial A_{z1}} = [...] = \frac{\partial U}{\partial A_{zi}} = 0$$

Obtaining solution

k - global node index $[1..n]$

$$\frac{\partial U}{\partial A_{zk}} = M_k \sum_{j=1}^{n} A_{zj} C_{kj} + J_k = 0$$

Reorder node names so that unknown potential values have indexes $[1..m]$ and known have indexes $[m+1..n]$

$$\sum_{j=1}^{m} A_{zj} C_{kj} = -\sum_{j=m+1}^{n} A_{zj} C_{kj} - \frac{J_k}{M_k}$$

Can be viewed as matrix equation:
$Ax = B$

Obtaining solution

Eigen library has *solver* concept that either solves linear equation system or returns an error

Has several solvers that are tailored to different problems, e.g.

SimplicialLDLT – for medium sparse problems (2D Poisson)

ConjugateGradient – for large symmetric problems (3D Poisson)

SparseQR – for least squares problems

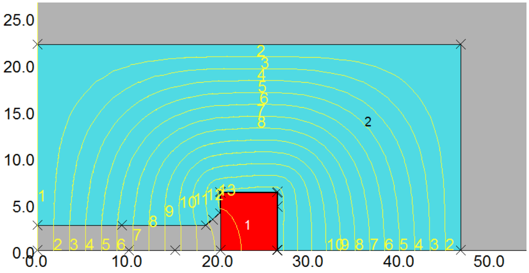
If solution exists, such solver would return all node values for magnetic potential

Obtaining solution

```
//define a solver
 Eigen::SimplicialLDLT<Eigen::SparseMatrix<double> > solver;
solver.analyzePattern(A);   // checks for non-zero values
solver.factorize(A);
if(solver.info()!=Success) {
  // decomposition failed
  return;
}
x1 = solver.solve(b1);
if(solver.info()!=Success) {
  // solving failed
  return;
}
x2 = solver.solve(b2); //can be used for different right parts
```

Obtaining solution

After solving linear equation system, we get all node values and can also calculate magnetic field



Summary

Using 2D FEM calculations, we can speed up magnets design in early stage, and people are working in almost real time.

One single calculation with 500x500 nodes takes fractions of seconds, while full 3D simulation could take several days.

Summary

To calculate fields using finite element method:

1. Generate triangle mesh for problem domain
2. Construct linear interpolation functions for each triangle
3. Express total energy of system in terms of interpolation functions and node values
4. Calculate integrals or differentials of interpolation functions
5. At this step energy is written as polynomial function of node values
6. Apply principle of minimal energy (minimize energy with respect to each unknown node value)
7. Compose system of linear equations (with unknown node values)
8. Solve resulting linear equation system