Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

Numerical Software II

Sparse Linear Algebra with Eigen

Viktor Mosenkis

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen University

Contents

Outline

Objectives and Learning Outcomes

Objective

- Introduction to sparse linear algebra,
- Use Eigen for sparse linear algebra.

Learning Outcomes

- You will understand
  - different formats (CRS, CCS) for sparse matrices,
  - their advantages and disadvantages,
  - sparse linear algebra with Eigen.
- You will be able to
  - solve sparse system of linear equations with Eigen.
  - solve sparse linear regression problem with Eigen.

Outline

Memory size for $n \times n$ matrix with **double** entries stored as 2D-array

- $n = 10^2$: memory used $10^4 \cdot$**sizeof(double)**$= 80\text{KB}$
- $n = 10^3$: memory used $10^6 \cdot$**sizeof(double)**$= 8000\text{KB} = 8\text{MB}$
- $n = 10^4$: memory used $10^8 \cdot$**sizeof(double)**$= 800\text{MB}$
- $n = 10^5$: memory used $10^{10} \cdot$**sizeof(double)**$= 8000\text{MB} = 8\text{GB}$
- $n = 10^6$: memory used $10^{12} \cdot$**sizeof(double)**$= 800\text{GB}$

How can we perform computations with big matrices?

In scientific computing a sparse matrix is a matrix in which most of the elements are zero. A matrix where most of the elements are nonzero, considered dense.

Sparsity is the proportion of the number of zero-valued elements in the matrix to the total number of elements. The matrix is considered to be sparse if its sparsity is greater than 0.5.

In scientific application large sparse matrices typically appear when solving partial differential equations.

For sparse matrices we could simply store only the nonzero elements!

Solve the discretized form of Laplace's equation,

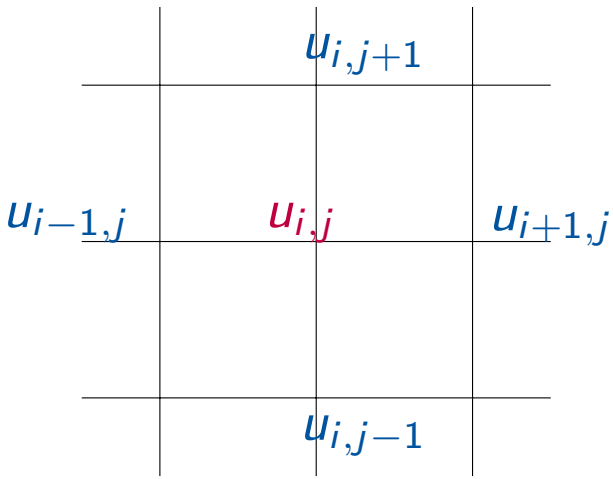
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad ,$$

for $u(x, y)$ defined within the domain of $0 \leq x \leq 1$ and $0 \leq y \leq 1$ given the boundary conditions

- $u(x, 0) = 1$
- $u(x, 1) = 2$
- $u(0, y) = 1$
- $u(1, y) = 2$

We now approximate the second partial derivatives in the PDE by 2nd order centered finite difference scheme,

- $\left(\frac{\partial^2 u}{\partial x^2}\right)_{i,j} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2}$
- $\left(\frac{\partial^2 u}{\partial y^2}\right)_{i,j} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2}$



Inserting the equations from the previous slide into the Laplace's equation at the grid point (i, j), yield

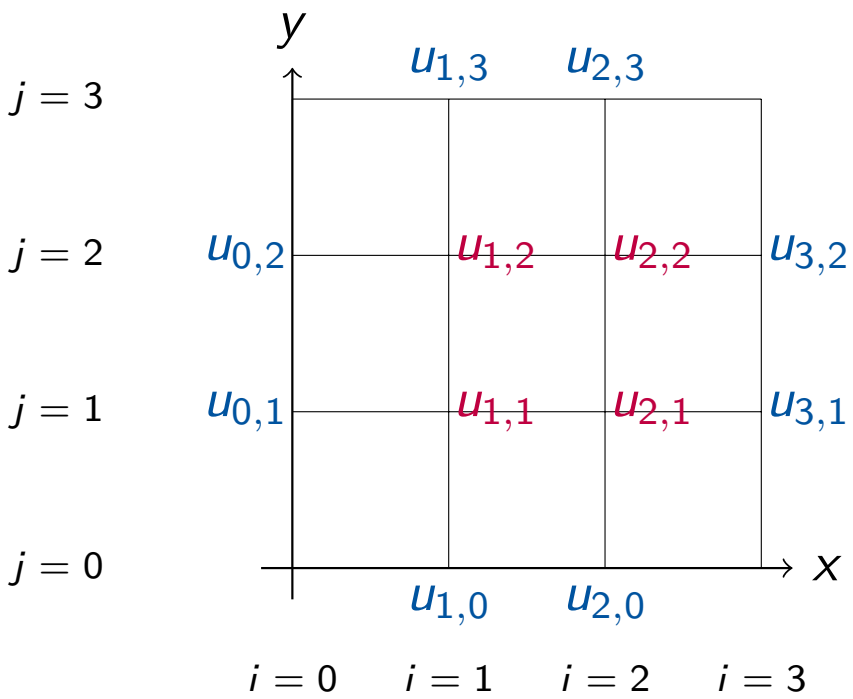
$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{(\Delta x)^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{(\Delta y)^2} = 0 \quad .$$

For $\Delta x = \Delta y$ we obtain

$$-4u_{i,j} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = 0$$

Hence the partial derivatives $\left(\frac{\partial^2 u}{\partial x^2}\right) + \left(\frac{\partial^2 u}{\partial y^2}\right)$, at the grid point (i, j) can be evaluated using the discrete values of u at (i, j) itself and the values at its four neighboring points. Write the equation for all interior points $u_{i,j}$ yield an equation system with at most five entries in each row. Hence the matrix is sparse.

Let's consider a small example of the discretization with just few grid points



with $u_{i,j} = u(i\Delta x, j\Delta y)$ and $\Delta x = \Delta y = 1/3$. The values of the variables $u_{i,j}$ in blue are given by the boundary conditions.

We can now write the equations for the four interior points $u_{i,j}$

$$\begin{array}{rrrrrrrrrrrr}
-4 & u_{1,1} & + & u_{1,2} & & & + & u_{2,1} & +u_{0,1} + u_{1,0} & = 0 \\
 & u_{1,1} & -4 & u_{1,2} & + & u_{2,2} & & & +u_{0,2} + u_{1,3} & = 0 \\
 & & & u_{1,2} & -4 & u_{2,2} & + & u_{2,1} & +u_{2,3} + u_{3,2} & = 0 \\
 & u_{1,1} & & & + & u_{2,2} & -4 & u_{2,1} & +u_{2,0} + u_{3,1} & = 0
\end{array}$$

The $u_{i,j}$ marked with blue are given by the boundary conditions. Bringing these values to the right hand side yield the following equation system

$$\begin{pmatrix} -4 & 1 & 0 & 1 \\ 1 & -4 & 1 & 0 \\ 0 & 1 & -4 & 1 \\ 1 & 0 & 1 & -4 \end{pmatrix} \begin{pmatrix} u_{1,1} \\ u_{1,2} \\ u_{2,2} \\ u_{2,1} \end{pmatrix} = \begin{pmatrix} -2 \\ -3 \\ -4 \\ -3 \end{pmatrix}$$

Outline

The Compressed Row Storage (CRS) and Compressed Column Storage (CCS) are general storage formats. They make absolutely no assumptions about the sparsity structure of the matrix and they don't store any unnecessary elements. The matrix is represented with the following three vectors

- Value vector (floating-point, size - number of nonzeros (nnz))
- InnerIndices vector (integer, size - nnz)
- OuterStarts vector (integer, size - number of rows $+1$ (CRS) number columns $+1$ (CCS))

CRS/CCS format puts the subsequent nonzeros of the matrix rows/columns in contiguous memory locations. The Value vector stores the nonzero elements of the matrix A. The InnerIndices vector stores the column/row indexes of the elements of the Value vector. That is if $\text{Value}(k) = a_{ij}$ then $\text{InnerIndices}(k) = j/\text{InnerIndices}(k) = i$. The OuterStarts vectors stores the locations in Value that start a row/column. By convetion the last entry of OuterStarts vectors is set to nnz.

CRS and CCS Formats

Example

Given the matrix

$$A = \begin{pmatrix} 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 0 & 0 & 17 \\ 7 & 5 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 8 \end{pmatrix}$$

Then

- Its CRS representation:

Values:	3	22	17	7	5	1	14	8
InnerIndices:	1	0	4	0	1	3	2	4
OuterStarts:	0	1	3	6	6	8		

- Its CCS representation:

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		

- Random access is expensive. We first need to access the OuterStarts vector to find the begining of the desired row/column and then search for the index of the desired value in the InnerIndices vector. So in worst case we will have to check all indices in the desired row/column. Leading to worst case complexity of $O(\text{row size})$/ $O(\text{column size})$ and not $O(1)$ as in dense case.

- Inserting a new nonzero element is expensive as we must reallocate the Values and InneIndices vector.

- Deleting an element is expensive if we want to free the memory. E.g. once it became zero as a result of some operation.

Fill-in of a matrix are those entries that change from an intitial zero to a nonzero value during execution of an algorithm.

$$\begin{pmatrix} * & * & * \\ * & * & 0 \\ 0 & * & 0 \end{pmatrix} \xrightarrow{LU} \begin{pmatrix} * & * & * \\ * & * & * \\ 0 & * & * \end{pmatrix} \quad \text{better} \quad \begin{pmatrix} * & * & 0 \\ 0 & * & 0 \\ * & * & * \end{pmatrix} \xrightarrow{LU} \begin{pmatrix} * & * & 0 \\ 0 & * & 0 \\ * & * & * \end{pmatrix}$$

Mimimizing fill-in by switching rows and columns in the matrix is important for several reasons

- reduce memory requirements
- avoid unnecessary reallocations of memory
- reduce the number of performed operations

Outline

Eigen's main sparse matrix representation is given in the class SparseMatrix. It implements a variant of CRS/CCS scheme and consists of four compact arrays.

- Value vector (floating-point, size - number of nonzeros (*nnz*))
- InnerIndices vector (integer, size - *nnz*)
- OuterStarts vector (integer, size - number of rows +1 (CRS) number columns +1 (CCS))
- InnerNNZs vector (integer, number of rows (CRS) number columns (CCS))

The vectors Value, InnerIndices, OuterStarts serve exactly the same purpose as in CRS/CCS formats, with the only difference, that Value and InnerIndices contain spare entries to quickly insert new elements. That is why the vector InnerNNZs is required to identify the actual number *nnz* in row/column.

Given the matrix

$$A = \begin{pmatrix} 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 0 & 0 & 17 \\ 7 & 5 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 8 \end{pmatrix}$$

Possible Eigen CCS representation:

Values:	22	7	◇	3	5	14	◇	◇	1	◇	17	8
InnerIndices:	1	2	◇	0	2	4	◇	◇	2	◇	1	4
OuterStarts:	0	3	5	8	10	12						
InnerNNZs:	2	2	1	1	2							

Due to the special storage scheme of SparseMatrix adding a new nonzero entry to the matrix has the cost of $O(nnz)$. The simplest way to create a sparse matrix while guaranteeing good performance is to build a list of triplets and then convert this list to SparseMatrix.

```
1  typedef Eigen::Triplet<double> T;
2  std::vector<T> tripletList;
3  tripletList.reserve(estimation_of_entries);
4  for(...)
5  {
6    // ...
7    tripletList.push_back(T(i,j,v_ij));
8  }
9  SparseMatrixType mat(rows,cols);
10 mat.setFromTriplets(tripletList.begin(), tripletList.end());
```

Convert Dense matrix to Sparse

```
1  Eigen::SparseMatrix<double> A;
2  Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic> B;
3  ...
4  A = B.sparseView();
```

Convert Sparse matrix to Dense

```
1  Eigen::SparseMatrix<double> A;
2  Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic> B;
3  ...
4  B = A;
```

Any SparseMatrix can be turned into proper CRS/CCS format by calling SparseMatrix::makeCompressed(). In this case the InnerNNZs vector is redundant and thus this call frees this buffer.

Solving Sparse Linear Equations

LU Decompositon


```
1  void solve_LES(const SparseMatrix<double> &A,
2    const Matrix<double,Dynamic,1> &b,
3    Matrix<double,Dynamic,1> &x)
4  {
5    SparseLU<SparseMatrix<double>, COLAMDOrdering<int> > solver;
6    solver.compute(A);
7    if (solver.info() != Success) std::cout << "Decomposition failed " << std::endl;
8    x = solver.solve(b);
9    if (solver.info() != Success) std::cout << "solver failed " << std::endl;
10 }
11
12 int main(int argc, char* argv[]) {
13   assert(argc==2); int n=std::stoi(argv[1]);
14   std::vector<Triplet<double>> coefficients;
15   Eigen::Matrix<double,Dynamic,1> b(n), x(n);
16
17   buildProblem(coefficients, b, n);
18   Eigen::SparseMatrix<double> A(n,n);
19   A.setFromTriplets(coefficients.begin(), coefficients.end());
20   solve_LES(A,b,x);
21 }
```

Solving Sparse Problems

Linear regression with Sparse QR


```
1  void Regression_QR(const SparseMatrix<double> &A,
2    Matrix<double,Dynamic,1> &p,
3    const Matrix<double,Dynamic,1> &x)
4  {
5    SparseQR<SparseMatrix<double>, COLAMDOrdering<int> > solver;
6    solver.compute(A);
7    if (solver.info() != Success) std::cout << "Decomposition failed " << std::endl;
8    p = solver.solve(x);
9    if (solver.info() != Success) std::cout << "solver failed " << std::endl;
10 }
11
12 int main(int argc, char* argv[]) {
13   assert(argc==3); int m=std::stoi(argv[1]); int n=std::stoi(argv[2]);
14   std::vector<Triplet<double>> coefficients;
15   Eigen::Matrix<double,Dynamic,1> p(n), y(n);
16   buildProblem(coefficients, y, m, n);
17   Eigen::SparseMatrix<double> A(m,n);
18   A.setFromTriplets(coefficients.begin(), coefficients.end());
19   Regression_QR(A,p,y);
20   std::cout << "||A*p-y|| = " << (A*p-y).squaredNorm() << std::endl;
21   std::cout << "||A^T*A*p-A^T*y||="
22             << (A.transpose()*A*p-A.transpose()*y).norm() << std::endl;
23 }
```

Outline

Summary

- Use sparse linear algebra with Eigen
  - CRS and CCS formats
  - Solve sparse systems of linear equations with Eigen
  - Solve sparse linear regression problem with Eigen

Next Steps

- Play with sample code.
- Compare performance of dense and sparse solvers for linear equation systems and linear regression implementations. Try wrappers to external solver mentioned in the Eigen documentation and compare the performance with built-in ones.
- Continue the course to find out more ...