Conjugate Gradient Method: Cost and Convergence

Also for the Conjugate Gradient Method the dominant computational cost is given by the matrix-vector product with A. Hence the cost is roughly $2n^2$ FLOPs per iteration.

Theorem 1

The Conjugate Gradient method converges in at most n iterations for all initial guess $\underline{x}^{(0)}$. Moreover it holds:

$$\left\|x - x^{(k)}\right\|_A \leq 2\left(\frac{\sqrt{\kappa_2(A)} - 1}{\sqrt{\kappa_2(A)} + 1}\right)^k \left\|x - x^{(0)}\right\|_A$$

where $\|v\|_A = \sqrt{v^T A v}$ is the $A-$norm.

Note that when A is SPD it holds

$$\kappa_2(A) = \|A\| \left\|A^{-1}\right\| = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$$

It follows that convergence is fast if $\lambda_{\max}(A) \approx \lambda_{\min}(A)$ (i.e. A is well-conditioned), or in other words if the eigenvalues are clustered. On the other hand, it might be slow if $\lambda_{\max}(A) \gg \lambda_{\min}(A)$ (i.e. A is ill-conditioned).

Preconditioners

To speedup the convergence, we can use a preconditioner. In this case, the original linear system $Ax = b$ with the equivalent one

$$P^{-1}Ax = P^{-1}b$$

where P is a nonsingular matrix called preconditioner.

A good preconditioner has two features:

- An iterative method applied to the new system should converge in less iterations than for the original system. This typically means that the eigenvalues of $P^{-1}A$ should be clustered (in the case of CG, this means $\lambda_{\max}(P^{-1}A) \approx \lambda_{\min}(P^{-1}A)$).

- At each iteration of an iterative method we need to compute a matrix-vector product with the system matrix. In the preconditioned case, this is done in two steps

  $$v \longrightarrow Av \longrightarrow P^{-1}(Av)$$

  Thus, computing matrix-vector products with P^{-1} (or equivalently solving a linear system with P) should be fast. Note that we never need to compute the matrix $P^{-1}A$ explicitly.

Preconditioners

- Hence a good preconditioner P should be as similar as possible to A, while being easy to invert:
- Let us consider two extreme cases: $P = I_n$ and $P = A$
  - If $P = A$, $P^{-1}A = I_n$ and any iterative method would converge in just 1 iteration (note that $\lambda_{\max}(P^{-1}A) = \lambda_{\min}(P^{-1}A) = 1$). On the other hand, applying P^{-1} is as difficult as solving the original system.
  - If $P = I_n$, then applying P^{-1} has no cost. On the other hand $P^{-1}A = A$, so there is no reduction in the number od iterations.
- A good preconditioner should find a balance between these two extremes.

Preconditioning for CG

- In general, the problem of finding a good preconditioner is very problem-specific.
- Some black-box preconditioners:
  - **Jacobi:** $P = \text{diag}(A)$.
  - **Symmetric Gauss-Seidel:** $P = L_* \text{diag}(A)^{-1} L_*^T$ where $L_* = \text{tril}(A)$.

Sparse Matrices

"A matrix is sparse if many of its coefficients are zero. The interest in sparsity arises because its exploitation can lead to enormous computational savings and because many large matrix problems that occur in practice are sparse."

- Page 1, Direct Methods for Sparse Matrices, 2nd Edition, 2017.

The **sparsity** of an $n \times n$ matrix A is

$$\frac{\text{nnz}(A)}{n^2}$$

where $\text{nnz}(A) = \#$ of nonzero entries in A. A matrix is **sparse** if its sparsity is $\ll 1$. A matrix that is not sparse is **dense**.

Sparse matrices

- Sparse matrices are extremely common in engineering and computer science. Some examples:
  - Network theory (e.g. social networks).
  - Data analysis and machine learning.
  - Discretization of differential equations.
  - ...
- Sparse matrices represent problems with possibly a large of number of variables, but where each variable "interacts" directly with few other variables (local interactions).

Formats of sparse matrices

- To save memory, it is convenient to store only the nonzero entries of a sparse matrix. There are several data structure that allow this.
- Compressed Sparse Column (CSC). The matrix A is specified by three arrays: `val`, `row_ind`, `col_ptr`, nrows.
  - `val` stores the nonzero entries of A, ordered from from top to bottom and left to right.
  - `row_ind` stores the row indices of the nonzero entries of A.
  - `col_ptr[i]` stores the index of the element in val that begins the first non-empty column starting from the i-th column.
  - `nrows` stores the number of rows.
  - `ncols` stores the number of cols.

  This is the format used by Matlab.
- Example:

$$A = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 8 & 2 \end{bmatrix} \qquad \begin{aligned} \texttt{val} &= \begin{bmatrix} 4 & 7 & 3 & 8 & 2 \end{bmatrix} \\ \texttt{row\_ind} &= \begin{bmatrix} 1 & 2 & 3 & 4 & 4 \end{bmatrix} \\ \texttt{col\_ptr} &= \begin{bmatrix} 1 & 3 & 3 & 5 \end{bmatrix} \\ \texttt{nrows} &= 4, \quad \texttt{ncols} = 4 \end{aligned}$$

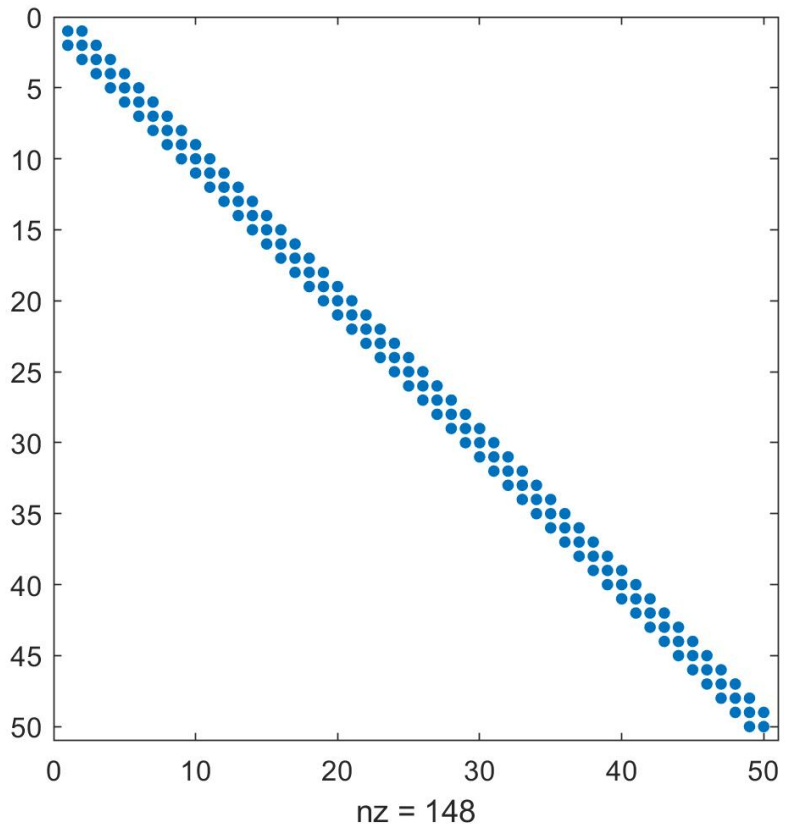
Sparsity and direct solvers

- We can take advantage of sparsity in Gaussian elimination, performing only the operations that are necessary (e.g. only the nonzero entries below a pivot are eliminated, and when summing two rows only the nonzero entries are summed). This allows to beat the $O(n^3)$ cost for dense matrices.
- This works particularly well in case of **banded matrices** (the nonzero are concentrated in a narrow "band" around the diagonal). In this case, a system can be solved in $O(n)$ operations.

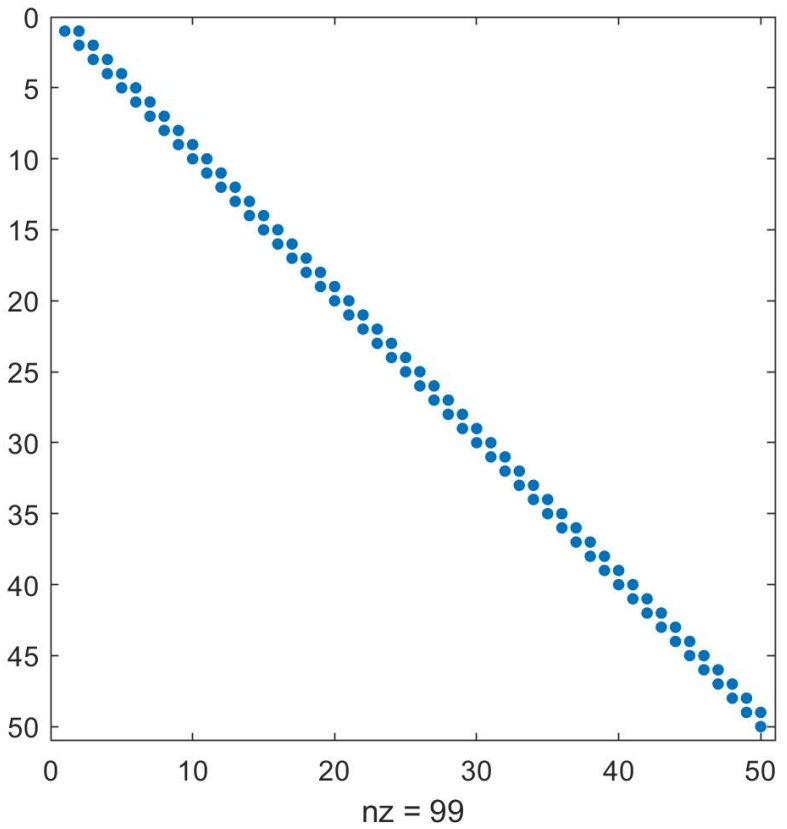
Example of a sparse matrix

A from 1D Poisson problem on $[0, 1]$ (discretized with FEM/FD, $h = 0.02$):

sparsity pattern of A



sparsity pattern of U



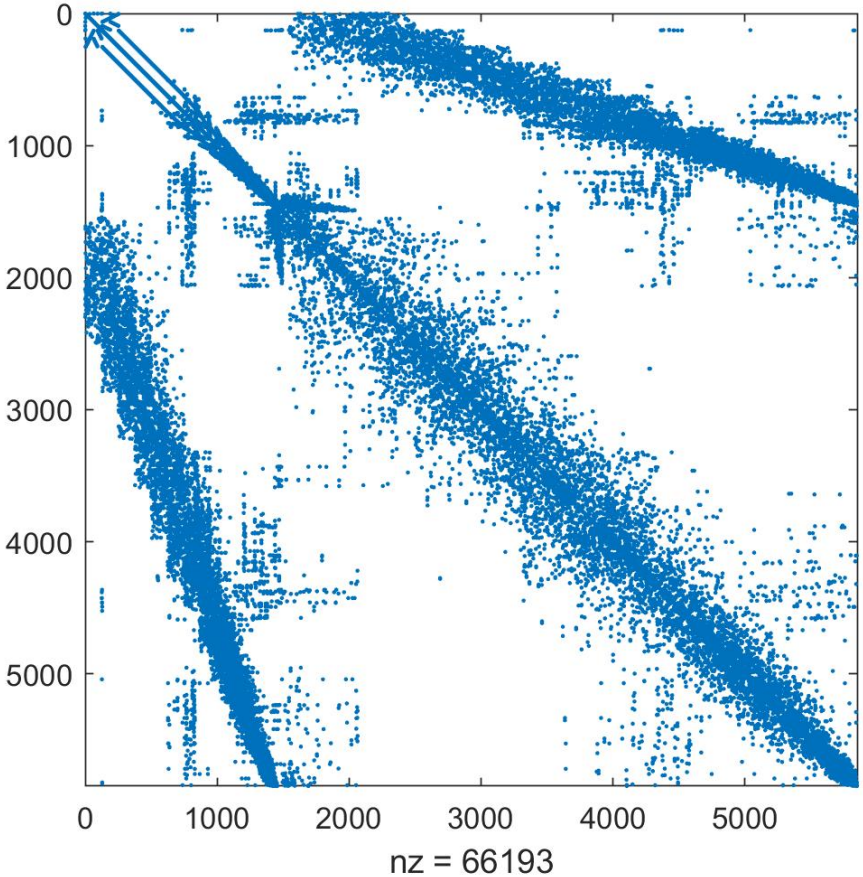
Fill-in

In general, the factor U (and L) can have much more nonzero entries than A. This phenomenon, known as **fill-in** significantly increases time and memory consumption, and represents the main drawback of direct solvers for sparse systems.

Example of fill-in

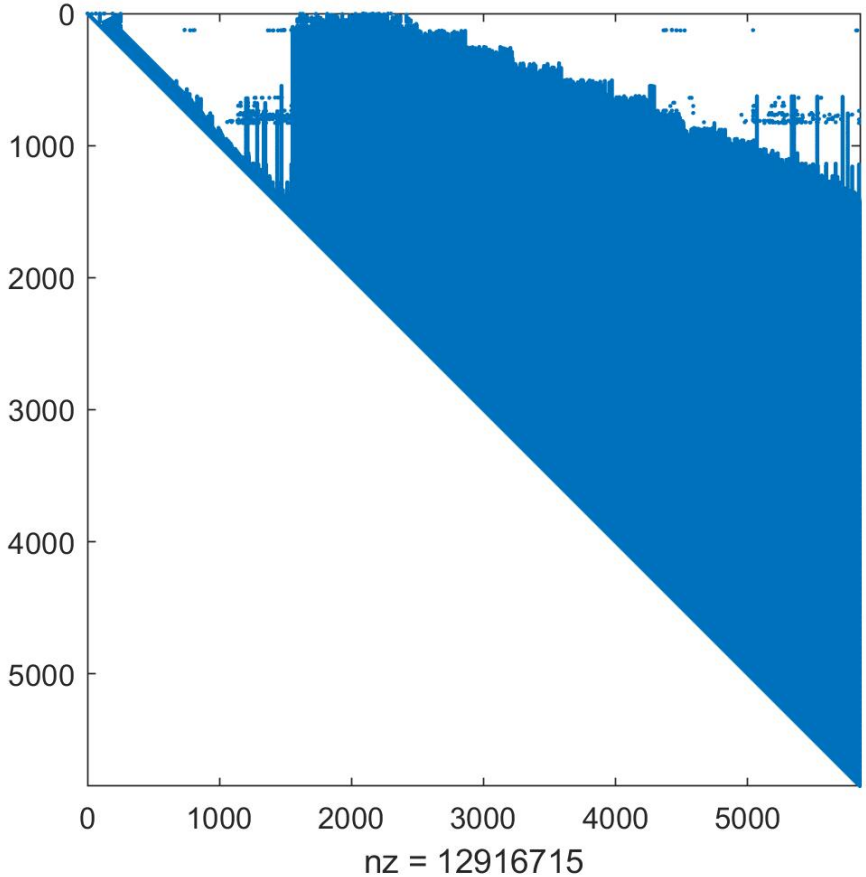
We consider a matrix A generated by a Finite Element Method for solving a 2D Poisson problem on the unit circle (meshsize $h = 0.05$).

sparsity pattern of A



memory(A) $\approx$ 1 MB

sparsity pattern of U



memory(U) $\approx$ 242 MB

Orderings

- The level of fill-in is often sensitive to the ordering of the variables
- Example:

$$A = \begin{bmatrix} x & x & x & x & x \\ x & x & & & \\ x & & x & & \\ x & & & x & \\ x & & & & x \end{bmatrix} \quad \Longrightarrow \quad U = \begin{bmatrix} x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ & & & & x \end{bmatrix}$$

  A is sparse but U is completely dense.

- But if we re-order rows and columns from last to first:

$$A = \begin{bmatrix} x & & & & x \\ & x & & & x \\ & & x & & x \\ & & & x & x \\ x & x & x & x & x \end{bmatrix} \quad \Longrightarrow \quad U = \begin{bmatrix} x & & & & x \\ & x & & & x \\ & & x & & x \\ & & & x & x \\ & & & & x \end{bmatrix}$$

  In this case, there is no fill-in.

Sparse matrices and iterative solvers

- Recall that at each iteration of an iterative method we have to compute a matrix-vector product

$$v \longrightarrow Av$$

If A is sparse, only the nonzero entries of A are involved in the computation:

$$(Av)_i = \sum_{j=1}^{n} a_{ij} v_j = \sum_{j \; s.t. \; a_{ij} \neq 0}^{n} a_{ij} v_j$$

The cost of a matrix-vector product is then $2\text{nnz}(A)$, versus $2n^2$ for dense matrices.

- Iterative solvers do not suffer from fill-in. In particular, the main memory consumption is just the storing of A. Hence iterative methods typically require much less memory than direct methods.

Summary on Linear Systems

Let us summarize the methods available for solving $A\underline{x} = \underline{b}$, with $A \in \mathbb{R}^{n \times n}$ non singular.

Direct Methods

Methods	Requirements on A	Cost
GEM / LU	[1] $\det(A_i) \neq 0,\ i = 1, \ldots, n$	$\sim 2/3\ n^3$ FLOPs
GEM / LU + Pivoting	none	$\sim 2/3\ n^3$ FLOPs

[1] A_i is the matrix obtained considering only the first i rows and the first i columns of A. This condition is automatically satisfied if A is diagonally dominant or if A is SPD.

Iterative Methods ($\sim 2n^2$ FLOPs for each iteration)

Methods	Requirements on A	Sufficient conditions for convergence
Jacobi	$A_{ii} \neq 0,\ i = 1, \ldots, n$	A diagonally dominant
Gauss-Seidel	$A_{ii} \neq 0,\ i = 1, \ldots, n$	A diagonally dominant or A SPD
Steepest Descent Method	A SPD	always ensured
Conjugate Gradient Method	A SPD	always ensured in less than n iterations