Lecture 3

Iterative methods for solving of linear systems of equations

outline

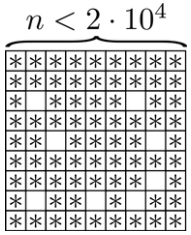
- dense and sparse matrices
- sparse matrix storage formats: COO, CSR
- simple iterative methods: Jacobi, Gauss-Seidel, overrelaxation
- directional iterative methods: Steepest Decent, Conjugate Gradients
- numerical examples

Additional bibliography:

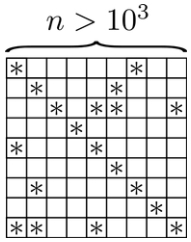
Yousef Saad, „*Iterative methods for sparse linear systems”*

R. Barnett et al., „Templates for the solution of Linear Systems: Building Blocks for Iterative Methods”

Dense and sparse matrices



- **dense matrix** – most of the matrix elements have nonzero value,
  more than 50% of all elements, SLE defined by
  such matrix can be efficently solved using the matrix factorization



- **sparse matrix** – most of the matrix elements are zeros and only small fraction
  of elements are nonzeros, usually much below 1-5% of all elements
  (example → tridiagonal matrix)

The use of standard factorization methods to sparse matrices have two severe drawbacks

- all elements are processed, including zeros,
  what is not neccessery and hence very inefficient (waste of time)

- more important is fact that sparse matrices very often have large sizes,
  number of columns/rows can reach n~10^5-10^6 !
  → such large matrices can not be put in computer memory !!!

For these reasons **iterative methods** for solving SLE defined by sparse matrices were developed,
their main features are as follows

- they operate on small number of matrix elements which are proccessed fast
  (matrix factorization is avoided)

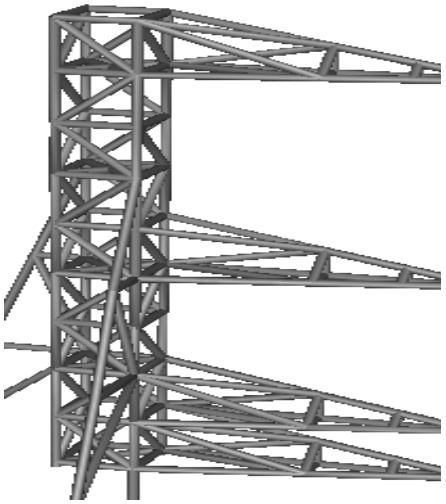
- save memory (only nonzero elements are retained)

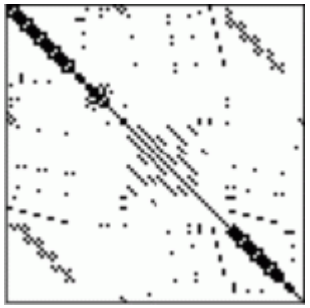
- unfortunately for badly conditioned matrices the iterative process might
  not converge to proper solution

Where we get the sparse matrices? Most often when we solve ODE/PDE problems with:
Finite Difference Method or **Finite Element Method**

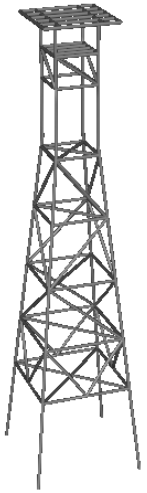
Examples from the **Matrix Market** → https://math.nist.gov/MatrixMarket/

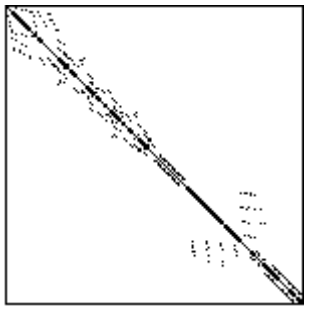
DWT 87 - tower



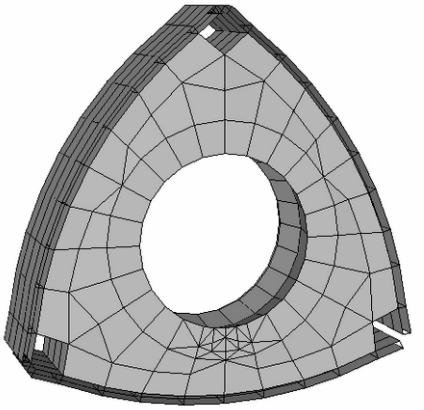


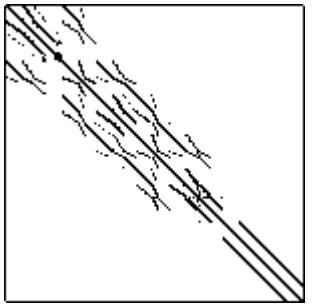
DWT 234: tower with platform





DWT 607: Wankel rotor





Sparse matrix storage formats

COO – coordinate format, matrix is represented by 3 vectors which retain:

- values of elements,
- indices of rows
- indices of columns

Remarks: (i) elements can be saved in any order
(ii) all information are stored explicitly

number of nonzeros: NNZ
memory (bytes) for double: NNZ*8 + 2*NNZ*4

CSR – compressed sparse rows, matrix is represented by 3 vectors:

- values of elements
- indices of columns
- global indices of the first elements appearing in subsequent rows

Remarks: (i) elements must be saved in row-wise order
(ii) row indices are encoded to save memory

number of nonzeros: NNZ
memory (bytes) for double: NNZ*8+NNZ*4+N*4 ← most row indices are reconstructed during computations

Other formats: CSC, diagonal, skyline, ... → see book of Saad or Matrix Market web page

Example: COO and CSR storage formats for sparse nonsymmetric and symmetric matrices

$$A = \begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -1 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -3 & 0 & 6 & 7 & 0 \\ 0 & 0 & 4 & 0 & -5 \end{bmatrix}$$

general/nonsymmetric matrix – all elements

COO:

```
NNZ  = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
vals = 1,-1,-3,-1, 5, 4, 6, 4,-3, 6,  7,  4, -5
cols = 0, 1, 3, 0, 1, 2, 3, 4, 0, 2,  3,  2,  4
rows = 0, 0, 0, 1, 1, 2, 2, 2, 3, 3,  3,  4,  4
```

CSR:

```
NNZ  = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
vals = 1,-1,-3,-1, 5, 4, 6, 4,-3, 6,  7,  4, -5
cols = 0, 1, 3, 0, 1, 2, 3, 4, 0, 2,  3,  2,  4
rows = 0, 3, 5, 8,11, NNZ
```

symmetric matrix – only lower or upper triangle matrix elements

CSR (upper)

```
NNZ  = 0, 1, 2, 3, 4, 5, 6, 7, 8
vals = 1,-1,-3, 5, 4, 6, 4, 7,-5
cols = 0, 1, 3, 1, 2, 3, 4, 3, 4
rows = 0, 3, 4, 7, 8, NNZ
```

Transformation beetwen COO & CSR formats can be made with **Intel-MKL** routine

```
call mkl_zcsrcoo (job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

as well as between dense matrix & CSR

```
call mkl_zdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

Matrix (CSR) – vector multiplication

The most demanding part of computations in iterative methods are the matrix-vector mulitplications and calculations of the vector scalar products.

For the matrix encoded in CSR format the matrix-vector multiplication is performed as follows

N- number of rows

$\vec{a}$ - vector of matrix elements

$\vec{c}$ - vector of column indices

$\vec{w}$ - vector of N indices of first elements in rows

nonsymmetric matrix

$$A\vec{x} = \vec{y} \qquad \Longrightarrow \qquad \sum_{j=1}^{n} a_{ij}x_j = y_i$$

```
y⃗ = 0⃗
for(i = 0; i < N; i++){
    for(l = w_i; l < w_{i+1}; l++){
        j = c_l  - column number
        y_i += a_l · x_j
    }
}
```

symmetric matrix

$$\sum_{j=1}^{n} a_{ij}x_j = y_i \rightarrow \sum_{j\geq i}^{n} a_{ij}x_j + \sum_{j<i} x_j a_{ji}$$

```
y⃗ = 0⃗
for(i = 0; i < N; i++){
    for(l = w_i; l < w_{i+1}; l++){
        j = c_l            - column number
        y_i += a_l · x_j    - first sum

        if(i != j) y_j += a_l · x_i  - second
                                       sum
    }
}
```

Jacobi method

Our task is to solve SLE with sparse matrix

$$A\vec{x} = \vec{b}, \qquad A \in R^{n\times n}, \quad \vec{x}, \vec{b} \in R^n$$

Any iterative method we start with some approximated vector x_0 which is then iteratively changed,
we hope that after many iterations (changes) this vector will be close to exact solution,
in other words the norm of residual vector shall be enough small to be numerically acceptable.

$$\vec{x}^{(0)} \longrightarrow \vec{x}^{(1)} \longrightarrow \vec{x}^{(2)} \longrightarrow \ldots \longrightarrow \vec{x}^{(p)} \qquad \Longrightarrow \qquad \|r^{(n)}\| = \|\vec{b} - A\vec{x}^{(n)}\| < \varepsilon ?$$

- often choice of x_0 → fill it with random numbers

(ε – small non-negative number)

Remark: in further consideration (Jacobi & Gauss-Seidel) we assume notation

- lower index enumerates elements in vector
- upper index stands for iteration number

$$\vec{x}^{(k)} = [\xi_1^{(k)}, \xi_2^{(k)}, \ldots, \xi_n^{(k)}]$$

$$\vec{b} = [\beta_1, \beta_2, \ldots, \beta_n]$$

According to the main assumption of iterative method we wish to bring all elements of residual vector to vanish.
In particular we may write this condition for the i-th element in the k-th iteration

$$\left(\vec{r}^{(k)}\right)_i = (\vec{b} - A\vec{x}^{(k)})_i = 0, \qquad i = 1, 2, \ldots, n$$

$$\beta_i - \sum_{j=1}^{n} a_{ij}\xi_j^{(k)} = 0$$

$$\vec{x}^{(k)} = [\xi_1^{(k)}, \xi_2^{(k)}, \ldots, \xi_n^{(k)}]$$

$$\vec{b} = [\beta_1, \beta_2, \ldots, \beta_n]$$

next rearrange the terms leaving the i-th on the left side

$$a_{ii}\xi_i^{(k)} = \beta_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}\xi_j^{(k)}$$

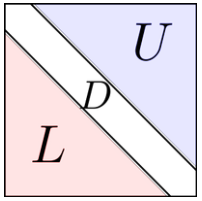
Because the elements of residual vector vanished in k-th iteration, it must be the truth also for next k+1 iteration what allows us to rewrite this expression accounting for this fact

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}}\left(\beta_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}\xi_j^{(k)}\right)$$

iteration formula in Jacobi method

By applying this equation to every element in vector x in present k-th iteration (right side) we get an improved (k+1)-th approximation (left side).

Jacobi iterative formula can be rewritten in more compact matrix form if we express the matrix as the sum of: the lower triangle, upper triangle and diagonal matrices



$$A = L + D + U$$

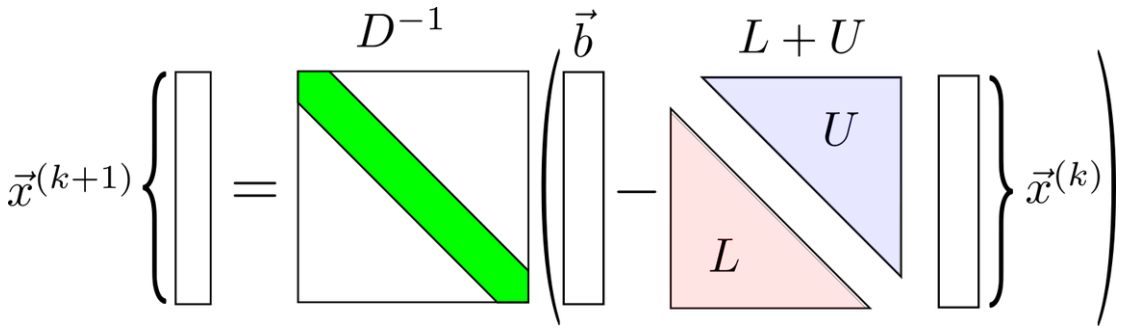
$$\vec{b} - A\vec{x}^{(k)} = 0$$

$$\vec{b} - (L + D + U)\vec{x}^{(k)} = 0$$

$$D\vec{x}^{(k)} = \vec{b} - (L + U)\vec{x}^{(k)}$$

Jacobi iteration formula in matrix form

$$\vec{x}^{(k+1)} = D^{-1}\left(\vec{b} - (L + U)\vec{x}^{(k)}\right)$$



Gauss-Seidel method

It is based on Jacobi method but with one essential modification – the elements already calculated in present iteration are immediately used to improve the remaining fraction of a vector.

To derive the iteration formula we start as in Jacobi method but divide the sum into two parts: improved elements (k+1) are multiplied by lower triangle matrix elements while the older (k) ones by the upper triangle matrix, respectively

$$\beta_i - \sum_{j=1}^{i-1} a_{ij}\xi_j^{(k+1)} - a_{ii}\xi_i^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k)} = 0$$

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}}\left(-\sum_{j=1}^{i-1} a_{ij}\xi_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k)} + \beta_i\right)$$

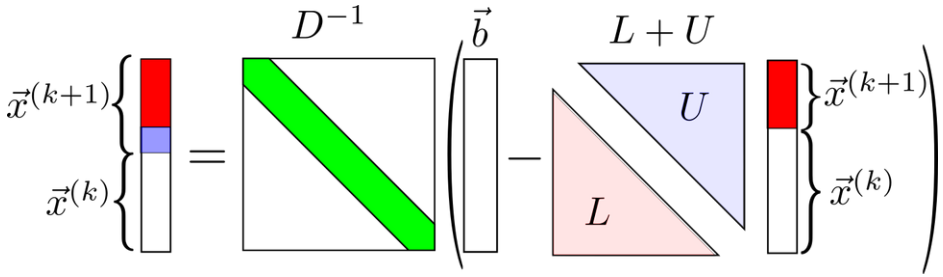
iteration formula of Gauss-Seidel

and analogoulsy as in Jacobi method we can express the formula in matrix form

$$A = L + D + U$$

$$\vec{b} - L\vec{x}^{(k+1)} - D\vec{x}^{(k+1)} - U\vec{x}^{(k)} = 0$$

$$\vec{x}^{(k+1)} = -D^{-1}L\vec{x}^{(k+1)} - D^{-1}U\vec{x}^{(k)} + D^{-1}\vec{b}$$



Remark: GS method is much faster than Jacobi method

Successive OverRelaxation – SOR method

For Jacobi and GS method we get iterative formulas

$$D\vec{x}^{(k+1)} = \vec{b} - (L+U)\vec{x}^{(k)}$$
$$(D+L)\vec{x}^{(k+1)} = \vec{b} - U\vec{x}^{(k)}$$

We see that shifting L to the left side improves efficiency with respect to Jacobi formula with just D on the left - corresponding to multiplication of L by zero.

This suggests that if we multiply L by the scaling factor we might eventually manipulate the pace of convergence. Scaling must be taken into account at factorization step

$$\begin{aligned} A &= L + D + U \\ \omega A &= \omega D + \omega L + \omega U + D - D \\ \omega A &= (D + \omega L) + (\omega U - (1-\omega)D) \end{aligned}$$

after some algebra we get

$$(D+\omega L)\vec{x}^{(k+1)} = [-\omega U + (1-\omega)D]\vec{x}_k + \omega\vec{b}, \qquad \omega \in [1,2]$$

and by rewriting it into dependence for particular element of vector x

$$\xi_i^{(k+1)} = \omega\xi_i^{(k+1)GS} + (1-\omega)\xi_i^{(k)}$$

The convergence factor ω mixes the old k-th contribution with the new one obtained from GS method, depending on its value we distinct three regimes of work

$$\begin{cases} \omega \in (0,1), & \text{subrelaxation} \\ \omega = 1, & \text{normal relaxation (GS-method)} \\ \omega \in (1,2), & \text{overrelaxation} \end{cases}$$

Steepest Descent method (SD) – symmetric & positively defined matrix

Besides iteration to fixed point methods (Jacobi/GS/SOR) we may use the directional methods to solve SLE. They search for better solution by moving along iteratively changed directions with steps scaled by a real number

from now to the end of lecture lower index enumerates iteration !!!

$$\vec{x}_0 \to \vec{x}_1 \to \ldots \to \vec{x}_N$$
$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{v}_k$$

α_k - scaling factor

$\vec{v}_k$ - searching direction

When the matrix of SLE is positivly defined we may define the n-dimensional quadratic function which helps us find direction and scaling number

$$\vec{x}^T A\vec{x} > 0 \qquad \Longrightarrow \qquad Q(\vec{x}) = \frac{1}{2}\vec{x}^T A\vec{x} - \vec{x}^T\vec{b}$$

gradient of Q is one of possible choices for searching direction in next iteration

$$\nabla_{\vec{x}_k} Q = A\vec{x}_k - \vec{b} = -\vec{r}_k = \vec{v}_k$$

In order to find the value of scaling factor α we must find minimum of Q with respect to α in (k+1)-th iteration (α becomes the variational parameter)

$$Q(\vec{x}_{k+1}) = Q(\vec{x}_k + \alpha_k\vec{v}_k) \qquad \Longrightarrow \qquad \frac{dQ(\vec{x}_{k+1})}{d\alpha_k} = 0 \qquad \Longrightarrow \qquad \alpha_k = -\frac{\vec{r}_k^T\vec{r}_k}{\vec{r}_k^T A\vec{r}_k}$$

iterative formula for Steepest Descent method

$$\vec{x}_{k+1} = \vec{x}_k + \frac{\vec{r}_k^T\vec{r}_k}{\vec{r}_k^T A\vec{r}_k}\vec{r}_k \quad \Longrightarrow \quad Q(\vec{x}_{k+1}) < Q(\vec{x}_k)$$

Conjugate Gradients method (CG) – symmetric matrix

We again conduct the iterative process to improve the solution

$$\vec{x}_0 \to \vec{x}_1 \to \ldots \to \vec{x}_N$$
$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{v}_k$$

α_k - scaling factor

$\vec{v}_k$ - searching direction

We assume the set of linearly independent vectors v_i (directions) span the vector subspace and the basis vectors are A-orthogonal (**A-conjugated**)

$$V = \{\vec{v}_1, \vec{v}_2, \vec{v}_3, \ldots, \vec{v}_k\}$$

$$\vec{v}_i^T A \vec{v}_j \equiv \delta_{ij}\, \vec{v}_i^T A \vec{v}_j \qquad (\delta_{ij}\text{- Kroenecker delta})$$

we may use this vector basis to express the difference between exact and present k-th approximated solution

$$\Delta\vec{x} = \vec{x}_{exact} - \vec{x}_k = \sum_{j=1}^{k} \alpha_j \vec{v}_j \qquad \vec{v}_k^T A \cdot /$$

$$\vec{v}_k^T A(\vec{x}_{exact} - \vec{x}_k) = \sum_{j=1}^{k} \alpha_j (\vec{v}_k^T A \vec{v}_j)\delta_{j,k}$$

scaling factor →

$$\alpha_k = \frac{\vec{v}_k^T(\vec{b} - A\vec{x}_k)}{\vec{v}_k^T A \vec{v}_k}$$

In CG method we use auxiliary basis vectors which are the residual vectors

$$R = \{\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_k\}$$

these are given Gram-Schmidt A-orthogonalization to get the target vectors v_i

$$\vec{v}_1 = \vec{r}_1$$

$$\vec{v}_{k+1} = \vec{r}_{k+1} - \sum_{j=1}^{k} \beta_j \vec{v}_j \qquad \vec{v}_k^T A \cdot /$$

$$\beta_k = \frac{\vec{v}_k^T A \vec{r}_{k+1}}{\vec{v}_k^T A \vec{v}_k}$$

← orthogonalization is required for the last vector v_k only

- basic CG method requires 2 matrix-vector multiplications while the modified method needs only one (**red**)

basic CG algorithm

initialization:

ε, K_{max}, $k = 0$

$\vec{v}_0 = \vec{r}_0 = \vec{b} - A\vec{x}_0$

DO

$k++$

$\alpha_k = \frac{\vec{v}_k^T \vec{r}_k}{\vec{v}_k^T A \vec{v}_k}$

$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{v}_k$

$\vec{r}_{k+1} = \vec{r}_k - \alpha_k A \vec{v}_k$

$\beta_k = \frac{\vec{v}_k^T A \vec{r}_{k+1}}{\vec{v}_k^T A \vec{v}_k}$

$\vec{v}_{k+1} = \vec{r}_{k+1} - \beta_k \vec{v}_k$

WHILE $(k < K_{max}$ && $\|\vec{r}_k\| > \varepsilon)$

optimal CG algorithm

initialization:

ε, K_{max}, $k = 0$

$\vec{v}_0 = \vec{r}_0 = \vec{b} - A\vec{x}_0$

DO

$k++$

$\alpha_k = \frac{\vec{r}_k^T \vec{r}_k}{\vec{v}_k^T A \vec{v}_k}$

$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{v}_k$

$\vec{r}_{k+1} = \vec{r}_k - \alpha_k A \vec{v}_k$

$\beta_k = \frac{\vec{r}_{k+1}^T \vec{r}_{k+1}}{\vec{r}_k^T \vec{r}_k}$

$\vec{v}_{k+1} = \vec{r}_{k+1} + \beta_k \vec{v}_k$

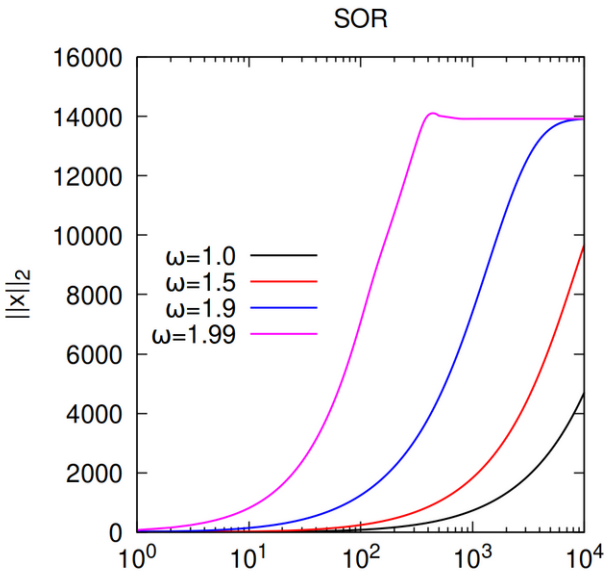
WHILE $(k < K_{max}$ && $\|\vec{r}_k\| > \varepsilon)$

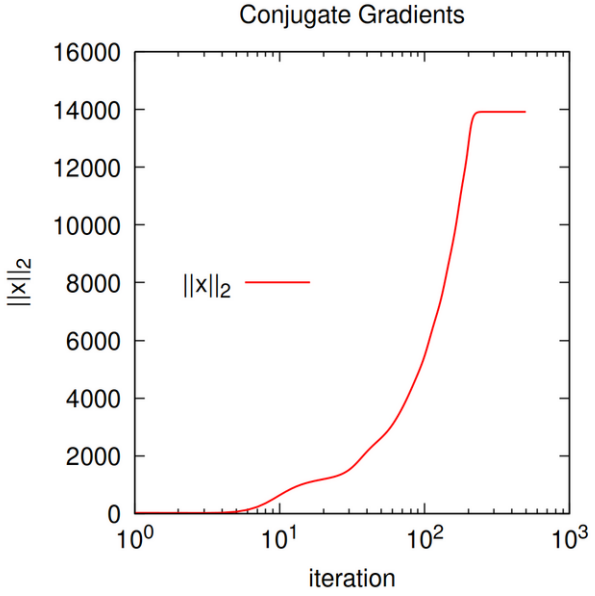
Example: solution of SLE with sparse matrix (tridiagonal) by SOR and CG methods

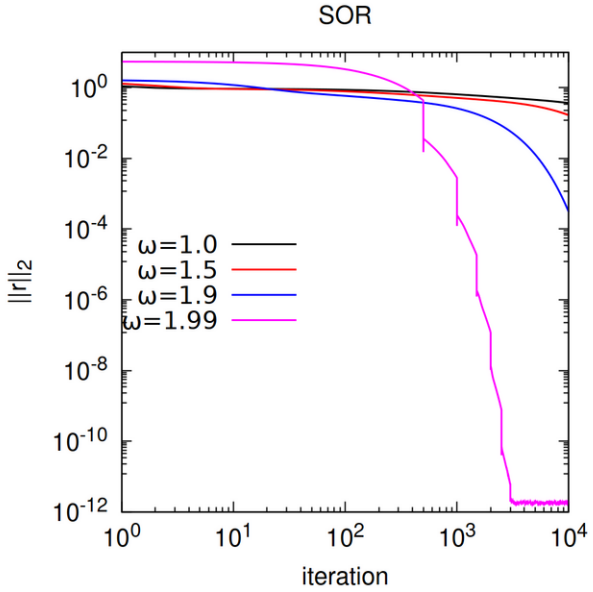
$$n = 500, \quad R^{n\times n} \ni A = [a_{ij}], \qquad a_{ij} = \begin{cases} \frac{-2}{1-3|i-j|}, & |i-j| \le 1 \\ 0, & |i-j| > 1 \end{cases}$$

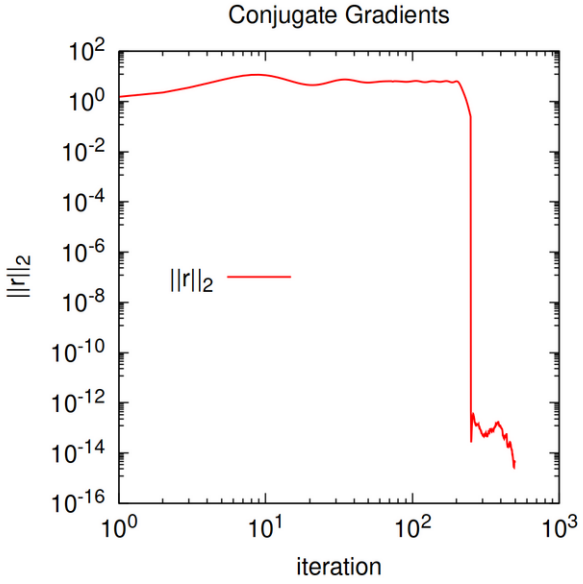
$$(\vec{b})_i = \frac{1}{10}\sin^{10}\left(\frac{2\pi i}{n-1}\right)$$

$$\vec{x}_0 = [1, 1, 1, \ldots, 1]^T$$









Remarks: (i) classification of iterative methods on account on efficency
(worst to best)

Jacobi – GS – SOR – SD – CG

(ii) SD and CG methods works for symmetric positively defined metrices

(iii) maximal number of steps in CG method equals n (usually much less is required)

(iv) for nonsymmetric matrices other iterative methods could be used

- Conjugate Gradient Square (**CGS**)
- Bi-Conjugate Gradent Stabilized (**BiCGStab**)
- General Minimal Residual Method (**GMRES**)
- Transpose-Free Quasi-Minimal Residual (**TFQMR**)
  more info → Saad book

(v) solving SLE with iterative method we can not predict how many iteration are required to find acceptable solution, therefore we shall assume the maximal number of iterations that can be performed to avoid infinite loop

(vi) the better starting vector x_0 we propose the less number of iteration is needed to get the solution

(vii) alternatively dedicated **sparse-matrix-LU factorization** can be applied, provided that we have enough memory, such factorization would require additional storage in memory at about 10-100(?) times larger than CSR storage of original matrix

freely-available packages:

PARDISO - Switzerland, provided with Intel MKL
MUMPS - France, see Linux repository, *https://mumps-solver.org/index.php*
SuperLU – USA, *https://portal.nersc.gov/project/sparse/superlu/*