Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Iterative Methods for Solving Linear Systems

Laboratory Work

Solving Sparse Linear Systems Using the Preconditioned Generalized Minimum Residual Method

Supported by Intel

K.A. Barkalov,
Software Department

Contents

- Problem Statement
- Generalized Minimum Residual Method
- Generalized Minimum Residual Method Implementation
- Preconditioned Generalized Minimum Residual Method
- Preconditioned Generalized Minimum Residual Method Implementation
- Experimental Results

Purposes of work

- Demonstrate practical implementation of the preconditioned generalized minimum residual method

Objectives of work

- Study the generalized minimum residual method
- Develop sequential implementation of the GMRes method for sparse matrices
- Method implementation using the ILU(0)-preconditioner
- Method convergence analysis. Comparison of method operation with and without preconditioner

Test infrastructure

CPU	Two Intel Xeon E5520 processors (4 core, 2.27 GHz)
RAM	16 Gb
OS	Microsoft Windows 7
Framework	Microsoft Visual Studio 2008
Compiler, profiler, debugger	Intel Parallel Studio XE 2011
Libraries	Intel® Threading Building Blocks 3.0 for Windows, Update 3 (part of Intel® Parallel Studio XE 2011)

Problem Statement (1)

- Let us consider a system of n linear equations like

$$Ax=b$$

- $A=(a_{ij})$ is a $n \times n$ real matrix; b and x are vectors consisting of n elements; let the exact system solution be x^*.
- *An iterative method* generates a sequence of vectors $x(s) \in R^m$, $s=0,1,2,\ldots$, where $x^{(s)}$ is an approximate system solution.
- An iterative method is *convergent* if

$$\forall\, x^{(0)} \in R^m \quad \lim_{s \to \infty} \left\| x^{(s)} - x^* \right\| = 0$$

- *Which iterative method stop criteria do you know?*

Problem Statement (2)

- Iterative method stop criteria: accuracy and number of iterations
  - Stop, if $\|x^{(s)}-x^{(s-1)}\| \le \varepsilon_1$. In this case, $\varepsilon_2=\|x^{(s)}-x^{(s-1)}\|$ is the attainable method accuracy.
  - Stop, if $\|r^{(s)}\| = \|Ax^{(s)} - b\| \le \varepsilon_1$. In this case, $\varepsilon_2=\|r^{(s)}\|$ is the attainable method accuracy.
  - Stop, if $s=N$. $x^{(N)}$ is understood as an obtained solution. The maximum number of iterations N is predefined.
- From this on, let us suppose that A is a real square matrix.

GENERALIZED MINIMUM RESIDUAL (GMRES) METHOD

Krylov Subspace Iterative Methods

- The Krylov subspace of dimension *m* generated by the vector *v* and matrix A is the linear space

$$K_m = K_m(A, v) = span\left\{v, Av, A^2v, ..., A^{m-1}v\right\}$$

- Having constructed the Krylov subspace, we can project the initi al system to it; the solution of the new system will consist in appr oximation to the initial system solution.
- Krylov subspace iterative methods create the subspace basis $K_m(A, v)$
- For the methods $v = r_0 / \| r_0 \|$ is usually selected, where r_0 is the first approximation residual x_0.

$$r_0 = b - Ax_0$$

Arnoldi orthogonalization (1)

- The objective is to find the orthogonal basis K_m, i. e. vectors $\{v_1, v_2, \ldots, v_m\}$
- Parameters: initial vector v_1 ($||v_1||=1$) and dimension m.

Algorithm

for $j = 1, ..., m$ do

$w = Av_j$ //next vector

for $i = 1, \ldots, j$ do

$h_{ij} = (w, v_i)$

$w = w - h_{ij} v_i$

end i

$h_{j+1,j} = ||w||$; $v_{j+1} = w / h_{j+1,j}$

end j

Arnoldi orthogonalization (2)

- As a result of the Arnoldi algorithm, we obtain:

  $V_m=[v_1,v_2,\ldots, v_m]$ is the orthonormal basis of the subspace K_m.

  $\overline{V}_m = [v_1, v_2,..., , v_m, v_{m+1}]$ – matrix expanded by the last computed vector v_{m+1}.

  H_m – orthogonalization coefficient matrix.

$$H_m = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} & \cdots & h_{1m} \\ h_{21} & h_{22} & h_{23} & h_{24} & \cdots & h_{2m} \\ 0 & h_{32} & h_{33} & h_{34} & \cdots & h_{3m} \\ \vdots & & \ddots & \ddots & & \vdots \\ 0 & \cdots & 0 & h_{m-1,m-2} & h_{m-1,m-1} & h_{m-1,m} \\ 0 & \cdots & 0 & 0 & h_{m,m-1} & h_{m,m} \end{bmatrix}$$

H_m – Hessenberg (upper) matrix

Arnoldi orthogonalization (3)

$\overline{H}_m$ – orthogonalization coefficient matrix expanded with a zero string with the coefficient $h_{m+1,m}$.

$$\overline{H}_m = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} & \dots & h_{1m} \\ h_{21} & h_{22} & h_{23} & h_{24} & \dots & h_{2m} \\ 0 & h_{32} & h_{33} & h_{34} & \dots & h_{3m} \\ \vdots & & \ddots & \ddots & & \vdots \\ 0 & \dots & 0 & h_{m-1,m-2} & h_{m-1,m-1} & h_{m-1,m} \\ 0 & \dots & 0 & 0 & h_{m,m-1} & h_{m,m} \\ 0 & \dots & & & 0 & h_{m+1,m} \end{bmatrix}$$

Generalized minimum residual (GMRes) method algorithm

1. Select as e_1 (first normalized basic vector)
   $v_1 = r_0 / \beta \quad (\beta = \| r_0 \| \ , \ r_0 = b - Ax_0)$.
2. Execute m steps of the Arnoldi algoritm (to obtain V_m, H_m)

for $j = 1, ..., m$ do

$w = Av_j$ //next vector

for $i = 1, \ldots, j$ do

$h_{ij} = (w, v_i)$

$w = w - h_{ij} v_i$

end i

$h_{j+1,j} = \|w\|; \ v_{j+1} = w / h_{j+1,j}$

end j

3. Compute $x_m = x_0 + V_m y_m$, where $y_m = \arg \min_y \left\| \beta e_1 - \overline{H}_m y \right\|$

GMRes – implementation (1)

- How to solve $y_m = \arg\min_y \left\| \beta e_1 - ?\overline{H}_m y \right\|$
- Equivalent formulation $\overline{H}_m y = \beta e_1$
  redefined linear system, matrix size $(m+1)\times m$.
  May be solved as a least squares problem.
- The matrix $\overline{H}_m$ is a Hessenberg one, simplify it to a triangle using Givens rotations.
- At the same time, r_i can be computed without computing x_i explicitly

GMRes – implementation (2)

- Let us define the rotation matrix Ω_i rotation coefficients χ_i, σ_i are selected so that $\eta_{i+1,i}$. is zeroed. At the same time, $c_i^2 + s_i^2 = 1$
- Multiply the system

$$\overline{H}_m y = \beta e_1$$

by the matrix Ω_i on the left.

$$\Omega_i = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & s_i & & & \\ & & & -s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix}$$

GMRes – implementation (3)

- Example:

$$\overline{H}_4 = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ & h_{32} & h_{33} & h_{34} \\ & & h_{43} & h_{44} \\ & & & h_{54} \end{bmatrix} \qquad \overline{g}_0 = \beta e_1 = \begin{bmatrix} \beta \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- Multiply by Ω_1, where

$$s_1 = \frac{h_{21}}{\sqrt{h_{11}^2 + h_{21}^2}} \qquad c_1 = \frac{h_{11}}{\sqrt{h_{11}^2 + h_{21}^2}}$$

$$\Omega_1 = \begin{bmatrix} c_1 & s_1 & & & \\ -s_1 & c_1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}$$

GMRes – implementation (4)

- Thus, we obtain

$$\overline{H}_4^{(1)} = \begin{bmatrix} h_{11}^{(1)} & h_{12}^{(1)} & h_{13}^{(1)} & h_{14}^{(1)} \\ & h_{22}^{(1)} & h_{23}^{(1)} & h_{24}^{(1)} \\ & h_{32} & h_{33} & h_{34} \\ & & h_{43} & h_{44} \\ & & & h_{54} \end{bmatrix} \qquad \overline{g}_1 = \begin{bmatrix} c_1\beta \\ -s_1\beta \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- Similarly, multiply by $\Omega_2, \Omega_3, \ldots$, to obtain

$$\overline{H}_4^{(4)} = \begin{bmatrix} h_{11}^{(4)} & h_{12}^{(4)} & h_{13}^{(4)} & h_{14}^{(4)} \\ & h_{22}^{(4)} & h_{23}^{(4)} & h_{24}^{(4)} \\ & & h_{33}^{(4)} & h_{34}^{(4)} \\ & & & h_{44}^{(4)} \\ & & & 0 \end{bmatrix} \qquad \overline{g}_4 = \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_5 \end{bmatrix}$$

GMRes – implementation (5)

- Set $Q_m = \Omega_m \cdot \Omega_{m-1} \cdot \ldots \cdot \Omega_1$, then

$$\overline{R}_m = \overline{H}_m^{(m)} = Q_m \overline{H}_m$$

$$\overline{g}_m = Q_m (\beta e_1) = (\gamma_1, \ldots, \gamma_m, \gamma_{m+1})^T$$

- As Q_m is a unitary matrix,

$$\min_y \left\| \beta e_1 - \overline{H}_m y \right\| = \min_y \left\| \overline{g}_m - \overline{R}_m y \right\|$$

- Let us delete the last line of the system and solve the triangular system

$$R_m y_m = g_m$$

for y.

GMRes – implementation (6)

- We can prove that:
  - The rank AV_m is equal to the rank R_m.
  - The vector $y_m = \arg\min_y \left\| \beta e_1 - \overline{H}_m y \right\|$ can be obtained by solving the system $R_m y_m = g_m$, i. e. $y_m = (R_m)^{-1} g_m$
  - The residual at the step m satisfies the following relations

$$r_m = b - Ax_m = \overline{V}_m [\beta e_1 - \overline{H}_m y_m] = \overline{V}_m Q_m^T (\gamma_{m+1} e_{m+1})$$

  - As a result, $\left\| r_m \right\| = \left| \gamma_{m+1} \right|$
- The indicated value may be used as a method stop criterion $\left\| r_m \right\| < \varepsilon$ and ensure such verification at each step of the algorithm.

SOFTWARE IMPLEMENTATION

Project creation (1)

- Create the 01_GMRes project to implement the generalized minimum residual method
- Include the following files in the project:
  - **types.h** – description of the involved data structures and invariables
  - **readMatrix.h, readMatrix.c** – declaration and implementation of the matrix reading functions from a mtx file
  - **memoryWork.h, memoryWork.c** – declaration and implementation of the functions working with memory
  - **matrixOperations.h, matrixOperations.c** – declaration and implementation of the functions for matrix-vector operations with sparse matrices

Project creation (2)

- **mklGMRes.h, mklGMRes.c** – declaration and implementation of the functions that solve systems using the FGMRes method from the Intel MKL library.
- **GMResSolver.h, GMResSolver.c** – declaration and implementation of the functions that solve linear systems using the generalized minimum residual method.
- **main.c** – call of the GMRes method and running serial experiments.

- Set the project to use the Intel MKL library: From **Intel C++ Composer XE 2011→Select Build Components,** select **Sequential** in the **Use MKL** field

Data structures

- For sparse matrix representation, use the CRS (Column Row Storage) format.
- In the file **types.h**, declare the structure **CrsMatrix:**

```
typedef struct CrsMatrix
{
  int N;                // matrix dimension (N x N)
  int NZ;               // number of nonzeroes
  FLOAT_TYPE* Value; // values array (dimension NZ)
  int* Col;             // column numbers array (NZ)
  int* RowIndex;        // row indices array (dimension N +1)
}
crsMatrix;
```

Main function

- Implement the **`main()`** function as follows:
  1. Enter the system parameters (name of the file with the matrix, required accuracy, maximum iterations number)
  2. Read the system matrix in the coordinate format
  3. Set the exact value $x_i = 1,\ i = 1, \ldots,\ n$, compute b
  4. Call the function for solving linear systems by the GMRes method, compute the achieved residual r norm
  5. Call the function for solving linear systems by the FGMRes method from the MKL library, compute the achieved residual r_{mkl} norm
  6. Display the results or save them in a file.

Auxiliary functions (1)

- Functions working with memory (**memoryWork.h, memoryWork.c**):
  - **InitializeMatrix()** – memory allocation to store a matrix in the CRS format. Input: dimension of the matrix **n**, number of nonzeroes **nz.** Output: pointer to the matrix storage structure **mtx.** The function returns the error code.

```
void InitializeMatrix(int N, int NZ, crsMatrix *mtx);
```

  - **FreeMatrix()** – memory release from the matrix in the CRS format. As an input, the function requires the pointer to the matrix storage structure **mtx.**

```
void FreeMatrix(crsMatrix *mtx);
```

Auxiliary functions (1)

- Matrix read function (**readMatrix.h, readMatrix.c**):
  - **ReadMatrixFromFile()** – reading the matrix from the file in the coordinate format, saving it in the CRS format. Input: name **matrixName** of the file containing a matrix Output: matrix dimension **n**, pointers to initialized arrays **column, row, val**, describing the matrix. The function returns the error code.

```
int ReadMatrixFromFile(char* matrixName, int* n, int**
column, int** row, double** val);
```

Auxiliary functions

- Matrix-vector operations (**matrixOperations.h, matrixOperations.c**):
  - **MultiplicateVV() –** vector scalar product computation. Input: pointers to the vectors **a, b**, their size **N**. Output: pointer to the scalar product.

```
void MultiplicateVV(const double* a, const double* b,
                    int N, double *c);
```

Auxiliary functions (3)

- **Mult_AlphaV_add_V()** – weighted vector total computation. Input: pointers to the vectors **a, b**, their dimension **N**, invariable multiplier **alpha.** Output: pointer to the vector **c**, where every **i** component is computed as **a[i] + b[i] * alpha.**

```
void Mult_AlphaV_add_V(const double* a, const double*
b, int N, double alpha, double* c);
```

- **MultVC()** – multiplication of all vector components by an invariable. Input: pointers to the vector **a**, its dimension **N**, invariable multiplier **alpha.** Output: pointer to the vector **c**, where every **i** component is computed as **a[i]*alpha.**

```
void MultVC (double *a, int N, double alpha, double*c);
```

Auxiliary functions (4)

- **MultiplicateMV() –** matrix-vector product computation. Input: pointers to the matrix **M** in the CRS format, vector **x**. Function output is the pointer to their product **b**.

```
void MultiplicateMV(const crsMatrix *A, const double*
x, double*b);
```

- **Normalize()** – vector normalization. Input: pointer to the vector **a**, its dimension **N.** Output: pointer to the vector **b** equal to the normalized vector **a.**

```
void Normalize(const double* a, int N, double* b);
```

Auxiliary functions (5)

- **InitializeJointRithtHandVector()** – setting the right-hand vector when the exact system solution - unit vector - is known. Input: pointers to the matrix A in the CRS format, right-hand vector b. Function output: initialized vector b.

```
void InitializeJointRithtHandVector(const crsMatrix
  *A, double* b);
```

- **ComputeResidualNorm() –** residual norm computation for the system with obtained solution. Input: pointers to the matrix A, right side vector b, solution vector res. Output: obtained residual norm value.

```
double ComputeResidualNorm(crsMatrix* A, double* res,
  double* b);
```

Auxiliary functions (6)

– **MultiplicateTransposedDenseMV()** – matrix-vector multiplication with a transposed matrix using the formula $x = alpha * aT * v + beta * x$, where **a** is a **n*m** matrix, **x** are vectors with a length **n**, **alpha** and **beta** are coefficients.

```
void MultiplicateTransposedDenseMV(double* a, double*
  v, int n, int m, double* x, double alpha,
  double beta);
```

Auxiliary functions (7)

– **UGaussSolve()** – solving the linear system using Gaussian method with a dense upper triangle matrix. Input: pointer to the matrix **A**, its number of rows **m**, number of columns **n**, right-hand vector **b**. The function returns the solution computed in the vector **x**.

```
double ComputeResidualNorm(crsMatrix* A, double* res,
                           double* b);
```

Givens rotations (1)

- Auxiliary function - **GivensRotations()** – performance of Givens rotations for the matrix and vector. Input: pointer to the **(m + 1)m** matrix **H**, number of columns **m**, vector **eBeta**, number of rotations **rotationsCount**, required computation accuracy **eps**. If the required accuracy is obtained earlier than **rotationsCount** steps, computation will be stopped and the number of rotations will be returned.

```
void GivensRotations(double* H, double* eBeta, int m,
                     int* rotationsCount, double eps);
```

Givens rotations (2)

```
void GivensRotations(double* H, double* eBeta, int m,
                     int* rotationsCount, double eps)
{
  ...

  for (i = 0; i < *rotationsCount; i++)
  {
    // invariables c, s
    c = H[i*m + i] / sqrt(H[i*m + i] * H[i*m + i] +
                H[(i + 1)*m + i] * H[(i + 1)*m + i]);
    s = H[(i + 1)*m + i] / sqrt(H[i*m + i] * H[i*m + i] +
                H[(i + 1)*m + i] * H[(i + 1)*m + i]);
    // matrix H modification
    for (j = i; j < *rotationsCount; j++)
    {
      oldValue = H[i*m + j];
      H[i*m + j] = oldValue * c + H[(i + 1)*m + j] * s;
      H[(i + 1)*m + j] =
                    H[(i + 1)*m + j] * c - oldValue * s;
    }...
```

Givens rotations (3)

```
      ...
      // right-hand vector eBeta modification
      oldValue = eBeta[i];
      eBeta[i] = c * oldValue;
      eBeta[i + 1] = -oldValue * s;

      // stop criterion verification
      if (fabs(eBeta[i + 1]) < eps)
      {
        *rotationsCount = i + 1;
        break;
      }
    }
  }
```

GMRes method implementation (1)

- Implement the function of solving the system by the generalized minimum residual method **GMResMethod().**
- Input: system matrix **A**, right-hand vector **b**, method parameter **M**, required method accuracy **eps**. The function returns the number of method iterations. Use the absolute residual norm of the system $||r^{(s)}|| = ||Ax^{(s)} - \beta|| \le \varepsilon_1$ as a method stop criterion.

```
int GMResMethod(const crsMatrix *A, const double* b,
                double *x, int m, double eps);
```

GMRes method implementation (2)

```
int GMResMethod(const crsMatrix *A, const double* b, double *x, int m,
                double eps)
{
  int i, j;
  int iterationsCount = m;
  // A->N * (m + 1) sized matrix v containing
  // Krylov subspace orthonormal basis
  // stored in a transposed form
  double* v = (double*)malloc(sizeof(double)*(m + 1)*A->N);
  double* w = (double*)malloc(sizeof(double)*A->N);
  // residual vector
  double* r0 = (double*)malloc(sizeof(double)*A->N);
  // auxiliary vector for the product A*x
  double* Ax0 = (double*)malloc(sizeof(double)*A->N);
  // Hessenberg matrix H sized m*(m + 1)
  double* H = (double*)malloc(sizeof(double)*m*(m + 1));
  // right-hand vector for a system containing Hessenberg matrix
  double* eBeta = (double*)malloc(sizeof(double)*(m + 1));
  ...
```

GMRes method implementation (3)

```
...
//1. Preliminary step
// computation of r[0] and its norm
MultiplicateMV(A, x, Ax0);
Mult_AlphaV_add_V(b, Ax0, A->N, -1.0, r0);
// computation of v[1]
Normalize(r0, A->N, v);
// matrix H initialization
memset(H,0,m*(m+1)*sizeof(double));
...
```

GMRes method implementation (4)

```
...
//2. Main iterations of the method
for (j = 0; j < m; j++)
{
  //computing w[j] = A * v[j]
  MultiplicateMV(A, v + A->N*j, w);

  for (i = 0; i <= j; i++)
  {
    //H[i][j] = (w[j], v[i])
    MultiplicateVV(w, v + A->N * i, A->N, H + i*m + j);
    //w[j] = w[j] - H[i][j]*v[i]
    Mult_AlphaV_add_V(w, v + A->N * i, A->N, -H[i*m + j],
                      w);
  }
  //H[j + 1][j] = ||w[j]||
  H[(j + 1)*m + j] = EuclideanNorm(w, A->N);
  ...
```

GMRes method implementation (5)

```
...
// has the exact solution been obtained?
  if (fabs(H[(j + 1)*m + j]) < EPSILON)
  {
    iterationsCount = j;
    break;
  }
  //v[j + 1] = w[j] / H[j + 1][j]
  MultVC(w, A->N, 1.0 / H[(j + 1)*m + j],
         v + A->N*(j + 1));
}
...
```

GMRes method implementation (6)

```
  ...
  //3. Approximated solution computation
  // right side initialization
  memset(eBeta,0,(iterationsCount + 1)*sizeof(double));
  eBeta[0] = EuclideanNorm(r0, A->N);
  // Givens rotation
  GivensRotations(H, eBeta, m, &iterationsCount, eps);
  // computing y[m] as the system solution
  // H*y = eBeta without the lower row
  UGaussSolve(H, eBeta, eBeta, iterationsCount, m);
  // x[m] computation, y[m] is stored in the vector eBeta
  MultiplicateTransposedDenseMV(v, eBeta, A->N,
    iterationsCount, x, 1.0, 1.0);

  // dynamic memory release
  ...
  return iterationsCount;
}
```

Use of the Intel MKL iterative solver. Interface (1)

- Intel MKL iterative solver - CG and FGMRes methods
- Connection:

```
#include "mkl_rci.h"    //iterative solver interface
#include "mkl_spblas.h"  //for auxiliary functions
```

- Main functions of the GMRes method:
  - Solver initialization

```
void dfgmres_init(MKL_INT *n, double *x, double *b,
                  MKL_INT *RCI_request, MKL_INT *ipar,
                  double *dpar, double *tmp);
```

  - Checking the set parameters for consistency

```
void dfgmres_check(MKL_INT *n, double *x, double *b,
                   MKL_INT *RCI_request, MKL_INT *ipar,
                   double *dpar, double *tmp);
```

Use of the Intel MKL iterative solver. Interface (2)

– Performance of a FGMRes iteration:

```
void dfgmres(MKL_INT *n, double *x, double *b,
             MKL_INT *RCI_request, MKL_INT *ipar,
             double *dpar, double *tmp);
```

- Parameters of functions:
  - **n, int** – system dimension
  - **x, double*** – current approximated system solution
  - **b, double*** – right-hand vector
  - **RCI_request, int** – current solver status
  - **ipar, int[128]** – array for the method integer parameters
  - **dpar, double[128]** – array for the method real parameters
  - **tmp, double*** – auxiliary array sized (2K + 1)*n + K(K + 9)/2 + 1, where K is the maximum allowed number of iterations that does not exceed the system order n

Use of the Intel MKL iterative solver. Interface (3)

- Some **ipar** values:
  - **ipar[4]** – maximum number of iterations (minimum from the system dimension and 150 by default)
  - **ipar[7]** – use of stop by iterations number criterion (0 - do not use, positive number - use*)
  - **ipar[8]** – use of stop by accuracy criterion (0 - do not use, positive number - use*)
  - **ipar[9]** – use of custom stop criterion (0 - do not use, positive number - use*)
  - **ipar[10]** – use of preconditioner (0 - do not use*, other - use)
  - **ipar[11]** – automatic computed orthogonal vector norm check (0 - do not use*, other - use) Here, * is the default value

Use of the Intel MKL iterative solver. Interface (4)

- Some **dpar** values:
  - **dpar[0]** – required relative accuracy (10^{-6} by default)
  - **dpar[1]** – required absolute accuracy (0 by default)

Use of the Intel MKL iterative solver. Interface

- The **RCI_request** status returned by **dfgmres()** takes on the following values:
  - 0 - desired solution found
  - 1 – multiply the system matrix A by a vector starting from tmp[ipar[21]-1] and save the result in the vector starting from tmp[ipar[22]-1]. Continue computation in **dfgmres().**
  - 2 - check the stop criterion. If the criterion is not met, continue computation in **dfgmres().**
  - 3 – apply the preconditioner to the vector starting from tmp[ipar[21]-1] and save the result in the vector starting from tmp[ipar[22]-1].
  - 4 – check the current orthogonal vector norm for zero
  - Negative value - the function has completed operation with an error.

Use of the Intel MKL iterative solver. Implementation (1)

```
int GMResMKL(crsMatrix* A, double* b, double *x, int maxIterations,
             double eps)
{
  int    ipar[128];  //parameter arrays
  double dpar[128];
  double *tmp;       // auxiliary array for solver operation
  int iterCount = 0; // number of performed iterations
  int RCI_request;   // solver opeation status
  char trans = 'N';  // variables for matrix-vector multiplication
  int i;
  //1. Data preparation
  // Array numbering customization A->Col, A->RowIndex (starting from
1)
  // memory allocation
  //2. Solver initialization
  dfgmres_init(&A->N, x, b, &RCI_request, ipar, dpar, tmp);
  if (RCI_request != 0)
  {
    MKL_Free_Buffers();
    return -1;
  } ...
```

Use of the Intel MKL iterative solver. Implementation (2)

```
//3. Parameters initialization
ipar[4] = maxIterations;
ipar[8] = 1; // use the accuracy stop criterion
ipar[9] = 0; // do not use the user stop criterion
ipar[11] = 1;// check the next approximation norm automatically
dpar[0] = 1.0e-15; // relative residual accuracy
dpar[1] = eps; // absolute residual accuracy

//4. Checking the set parameters
dfgmres_check(&A->N, x, b, &RCI_request, ipar, dpar,
              tmp);
if (RCI_request != 0)
{
  free(tmp);
  MKL_Free_Buffers();
  return -1;
}
...
```

Use of the Intel MKL iterative solver. Implementation (3)

```
//5. Method start
do
{
  dfgmres(&A->N, x, b, &RCI_request, ipar, dpar, tmp);
  // auxiliary vectors upgrade
  if (RCI_request == 1)
  {
    mkl_dcsrgemv(&trans, &A->N, A->Value, A->RowIndex,
     A->Col, &tmp[ipar[21]-1], &tmp[ipar[22]-1]);
  }
}
while(RCI_request != 0);

//6. Solution procedure
dfgmres_get(&A->N, x, b, &RCI_request, ipar, dpar, tmp, &iterCount);

//7. Completion of computation: memory release,
// Array indexing customization A->Col, A->RowIndex
return iterCount;
}
```

PRECONDITIONED GENERALIZED MINIMUM RESIDUAL METHOD

Idea of preconditioning

- $\mu_A=\lambda_{max}/\lambda_{min}$ is the spectral number.
  - $\mu_A \approx 1$ – the method converges quickly (A is well-conditioned)
  - $\mu_A >> 1$ – the method converges slowly (A is ill-conditioned)
- The idea of preconditioning lies in converting the ill-conditioned system $Ax=b$
  to a well-conditioned one

$$M^{-1}Ax=M^{-1}b.$$

  Here, M is a preconditioner.
- $M^{-1}A$ is not computed explicitly
- Corrective steps allowing for preconditioning are added to the iterative method.

Preconditioned GMRes method

1. Select as $e_1 = v_1$, where
   ($\beta = \| r_0 \|$, $r_0 = M^{-1}(b - Ax_0)$).
2. Execute m steps of the Arnoldi algoritms (to obtain V_m, H_m)

for $j = 1, ..., m$ do

$w = M^{-1}Av_j$ //next vector

for $i = 1, \ldots, j$ do

$h_{ij} = (w, v_i)$

$w = w - h_{ij} v_i$

end i

$h_{j+1,j} = \|w\|$; $v_{j+1} = w / h_{j+1,j}$

end j

3. Compute $x_m = x_0 + V_m y_m$, where $y_m = \arg \min_y \left\| \beta e_1 - \overline{H}_m y \right\|$

SOFTWARE IMPLEMENTATION OF THE PRECONDITIONED GMRES METHOD

Project creation (1)

- Create the 02_GMRes project to implement the preconditioned generalized minimum residual method
- Copy the files to the project from 01_GMRes
- Add the project with the ILU(0)-preconditioner from the respective laboratory work.
  - **ilu0.h, ilu0.c** – declaring and implementing functions of the ILU(0)-preconditioner
- Type of ILU(0) preconditioner:

$$M = LU$$

where L is the lower and U is the upper triangular matrix

Project creation (2)

- Set up the **02_PCGMethod** project properties: from **ConfigurationProperties**, in the **C\C++** → **General** →**Additional Include Directories** tab, enter **..\ILU** in the field **Additional Input Directories.**
- For the ILU project, indicate 01_GMRes in the property field.
- Connect Intel MKL to the project.
- Set up **Project Dependencies**: **ILU** depends on **01_GMRes** and **02_GMRes** depends on **ILU**.

Auxiliary functions

- Matrix-vector operations (**matrixOperations.h, matrixOperations.c**):
  - **GaussSolve()** – solution of linear systems with a dense upper or lower matrix. Input: pointers to the matrix **A**, system dimension **size**, right side vector **b**, number **isLowTriangle** denoting matrix type, i.e. upper triangle (0) or lower triangle (1). Output: pointer to the system solution **result**

```
void GaussSolve(crsMatrix* A, int size, int isLowTriangle,
                FLOAT_TYPE* b, FLOAT_TYPE* result);
```

  - This function is required when the preconditioned matrices L and U are used.

Auxiliary functions

– **LUmatrixSeparation()** – forming the multiplier matrices L and U based on the ILU-preconditioner matrix. As an input, the function receives the **ilu** preconditioner matrix and **uptr** array containing diagonal element indices in the Col array of the **ilu** matrix. The function returns structures of the matrices **L** and **U.**

```
void LUmatrixSeparation(crsMatrix* ilu, int *uptr,
                        crsMatrix* L, crsMatrix* U);
```

Main function

- Preconditioner call:

```
#include "ilu0.h "
int main( int argc, char* argv[])
{
  ...
  //4. Computing the matrices L and U
  ilu0(A->N, A->Value, A->Col, A->RowIndex, ILU->Value, uptr);
  LUmatrixSeparation(ILU, uptr, L, U);

  //5. Solving the system
  iterations = GMResMethod(A, b, x, L, U, M, eps);
  //7. MKL start
  iterationsMKL = GMResMKL(A, b, x, L, U, maxIterations, eps);
  finishMKL = clock();
  ...
}
```

Preconditioned GMRes implementation

```
int GMResMethod(const crsMatrix *A, const double* b, double *x,
   crsMatrix *L, crsMatrix *U, int m, double eps)
{ ...
  //1. Preliminary step - computation of r[0] and its norm
  MultiplicateMV(A, x, Ax0);
  Mult_AlphaV_add_V(b, Ax0, A->N, -1.0, r0);
  // use of preconditioner
  GaussSolve(L, A->N, 1, r0, buf);
  GaussSolve(U, A->N, 0, buf, r0);
  //2. Main iterations of the method
  for (j = 0; j < m; j++)
  {
    //computing w[j] = A * v[j]
    MultiplicateMV(A, v + A->N*j, w);
    // use of preconditioner
    GaussSolve(L, A->N, 1, w, buf);
    GaussSolve(U, A->N, 0, buf, w);
    ...
  } ...
}
```

Use of iterative solver of the Intel MKL library

```
int GMResMKL(crsMatrix* A, double* b, double *x, crsMatrix* L,
    crsMatrix* U, int maxIterations, double eps)
{ ...
  //3. Parameters initialization
  ipar[10] = 1;       // use a preconditioner
  //5. Method start
  do
  {
    dfgmres(&A->N, x, b, &RCI_request, ipar, dpar, tmp);
    // auxiliary vectors upgrade
    if (RCI_request == 1) { ... }
    // use of preconditioner
    else if (RCI_request == 3)
    {
      GaussSolve(L, A->N, 1, &tmp[ipar[21]-1], buf);
      GaussSolve(U, A->N, 0, buf, &tmp[ipar[22]-1]);
    }
  }
  while(RCI_request != 0); ...
}
```

EXPERIMENTAL RESULTS

Test matrices

- The University of Florida Sparse Matrix Collection
  http://www.cise.ufl.edu/research/sparse/matrices/
- Parameters of the matrices involved

Name	n	nz	μ_A
fs_183_1	183	998	21933900000000,00
fs_541_1	541	4282	4467,84
sherman2	1080	23094	964333000000,00
watt_1	1856	11360	4359640000,00
cage10	11397	150645	11,02

Experimental Results

- Method precision $\varepsilon = 10^{-4}$
- K – number of iterations, *Tolerance* – attainable solution accuracy

	Without preconditioning				With preconditioning			
	Own implementation		MKL		Own implementation		MKL	
Matrix	K	Tolerance	K	Tolerance	K	Tolerance	K	Tolerance
fs_183_1	57	1,242E-05	57	1,2414E-05	11	0,519593	10	2,04E-05
fs_541_1	9	1,943E-05	9	1,9435E-05	2	3,7E-11	2	3,62E-11
sherman2	100	20249205	100	117420734	12	423,1798	20	5,38E-05
watt_1	1	7,724E-07	1	7,7243E-07	32	1,19E-06	1	1,49E-07
cage10	14	6,987E-05	14	6,987E-05	4	3,92E-05	4	3,76E-05

Experimental Results

- Method precision $\varepsilon = 10^{-8}$
- K – number of iterations, *Tolerance* – attainable solution accuracy

	Without preconditioning				With preconditioning			
	Own implementation		MKL		Own implementation		MKL	
Matrix	K	Tolerance	K	Tolerance	K	Tolerance	K	Tolerance
fs_183_1	78	2.545E-06	59	5.7123E-07	16	0.00027	11	7.67E-07
fs_541_1	13	1.731E-09	13	1.7311E-09	2	3.7E-11	2	3.62E-11
sherman2	100	20249205	100	117420734	18	0.023276	22	1.37E-05
watt_1	40	9.876E-09	40	9.8761E-09	48	7.55E-11	9	9.84E-09
cage10	24	8.049E-09	24	8.0493E-09	7	9.55E-10	7	8.69E-10

Experimental Results

- Method precision $\varepsilon = 10^{-12}$
- K – number of iterations, *Tolerance* – attainable solution accuracy

	Without preconditioning				With preconditioning			
	Own implementation		MKL		Own implementation		MKL	
Matrix	K	Tolerance	K	Tolerance	K	Tolerance	K	Tolerance
fs_183_1	100	4.408E-06	59	5.7123E-07	31	3.31E-07	11	7.67E-07
fs_541_1	17	1.8E-13	17	1.85E-13	3	4.3E-14	3	4.9E-14
sherman2	100	20249205	100	117420734	24	0.000194	22	1.37E-05
watt_1	100	4.512E-10	100	1.293E-09	62	8E-15	35	5.99E-13
cage10	35	4.98E-13	35	4.93E-13	10	2.69E-13	31	1.77E-12

Experimental results. Conclusions

- For well-conditioned matrices
  - the method converges after 13 through 35 iterations depending on accuracy
  - If the accuracy grows 10^{-4} times, the number of iterations will grow by 10 iterations max.
- For ill-conditioned matrices,
  - the method converges after 40 through 80 iterations.
  - If the accuracy increases 10^{-4} times, the convergence rate will reduce.
- For most matrices, the use of preconditioner helps reduce the iterations count 3 through 5 times.

References

1. James W. Demmel. Applied Numerical Linear Algebra. SIAM, 1997.
2. Gene H. Golub, Charles F. Van Loan. Matrix Computations. The John Hopkins University Press, 1996.
3. J. Dongarra et al. Templates for the solution of linear systems: building blocks for iterative methods. SIAM, 1994.
4. Y. Saad. Iterative Methods for Sparse Linear Systems. SIAM, 2003.

Internet resources

5. Intel Math Kernel Library Reference Manual.

[http://software.intel.com/sites/products/documentation/hpc/mkl/mklman.pdf].