Lobachevsky State University of Nizhni Novgorod

Faculty of Computational mathematics and cybernetics

Iterative Methods for Solving Linear Systems

Laboratory Work
Solving Symmetric Sparse Linear Systems Using SOR Method with Chebyshev's Acceleration

Supported by Intel

K. A. Barkalov
Software Department

Contents

- ❑ Purpose and objectives of work
- ❑ Poisson equation of second order
- ❑ Computation scheme
- ❑ Successive Over Relaxation method
- ❑ Symmetric Successive Over Relaxation method
- ❑ Chebyshev’s acceleration of iterative processes
- ❑ Successive software implementation
- ❑ Method convergence analysis

Purposes of work

- ***The purpose of this work*** is to demonstrate acceleration of iterative methods by the example of the Symmetric Successive Over Relaxation method with Chebyshev's acceleration.

Objectives of work

- Studying the Successive Over Relaxation method to solve linear systems with general matrices.
- Development of the Successive Over Relaxation and Symmetric Successive Over Relaxation methods to solve linear systems with sparse matrices
- Development of the Symmetric Successive Over Relaxation method with Chebyshev's acceleration
- Development of infrastructure for mass experiments
- Development of successive implementation of the Symmetric Over Relaxation method with Chebyshev's acceleration to solve linear systems with sparse matrices.
- Developed method convergence analysis

Test infrastructure

CPU	No. 2 Intel Xeon E5520 (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)

POISSON EQUATION OF SECOND ORDER

Problem Statement (1)

Consider a Poisson equation of second order:

$$-\frac{\partial^2 u(x,y)}{\partial x^2}-\frac{\partial^2 u(x,y)}{\partial y^2}=f(x,y)$$

u - scalar twice differentiable function of two variables, x and y, considered for a square:

$$\{(x,y):0\le x,y\le 1\}$$

A Dirichlet problem is posed, which means that function values on the boundary of the region under consideration are set:

$$u(0,y)=u(1,y)=0 \quad 0\le y\le 1$$

$$u(x,0)=u(x,1)=0 \quad 0\le x\le 1$$

Problem Statement (2)

To be definite, set the required function as follows:

$$u(x, y) = \sin(\pi x) * \sin(\pi y)$$

To find numerical solution of a differential problem, a uniform square grid is introduced into the equation definition domain and the square approximation differencing scheme is used. The grid function v(x,y) which is the exact solution of the differencing scheme is treated as an approximate (numerical) solution of the initial problem.

SOR METHOD

Successive Over Relaxation (SOR) method

- Successive over relaxation method (SOR) is written as

$$\frac{(D+\omega L)(x^{(s+1)}-x^{(s)})}{\omega}+Ax^{(s)}=b$$

where ω is the method parameter.

- Convergence: $\omega \in (0,2)$ (required), if $A>0$, then it is sufficient
- For numerical solution of mathematical physics problems

$$\omega_{opt} \approx 2 - O(h)$$

- Required number of iterations when $\omega = \omega_{o\pi\tau}$: $O(h^{-1})$
  if $\omega = 1$ (for SOR it is the same as for the Seidel method): $O(h^{-2})$
- More accurate estimation

$$\omega_{opt} = \frac{2}{1+\sqrt{1-\rho^{2}\left(D^{-1}(R+L)\right)}}$$

SOR – algorithm

- ❑ With regard to $A–L=R+D$, let us put in a more convenient form

$$Dx^{(s+1)} = -\omega Lx^{(s+1)} + (1-\omega)Dx^{(s)} - \omega Rx^{(s)} + \omega b$$

- ❑ New approximation components are computed as

$$a_{ii}x_i^{(s+1)} = -\omega\sum_{j=1}^{i-1} a_{ij}x_j^{(s+1)} + (1-\omega)a_{ii}x_i^{(s)} - \omega\sum_{j=i+1}^{n} a_{ij}x_j^{(s)} + \omega b_i$$

- ❑ Iteration matrix $G_{SOR} = (D+\omega L)^{-1}((1-\omega)D - \omega R)$
  – non-symmetric!
- ❑ Total complexity of a single iteration

$$t_1=2n^2+n$$

- ❑ Performance of L iterations

$$T_1=L(2n^2+n).$$

SSOR – symmetric method

- A SSOR step consists of

1. A SOR step that involves computation of $x^{(s+1/2)}$ components in the normal order
2. A SOR step that involves computation of $x^{(s+1)}$ components in the reverse order

- SSOR step in a matrix form

1. $(D+\omega L)x^{(s+1/2)} = (1-\omega)Dx^{(s)} - \omega Rx^{(s)} + \omega b$
2. $(D+\omega U)x^{(s+1)} = (1-\omega)Dx^{(s+1/2)} - \omega Lx^{(s+1/2)} + \omega b$

- Iteration matrix

$$G_{SSOR} = (D+\omega U)^{-1}((1-\omega)D-\omega L)(D+\omega L)^{-1}((1-\omega)D-\omega R)$$

  - usually more iterations than for SOR with ω_{opt}
  - G_{SSOR} is a symmetric one, used for Chebyshev's acceleration

Chebyshev's acceleration

- ❑ Having found approximations $x^{(0)}, x^{(1)}, \dots, x^{(m)}$
- ❑ Let us find $y^{(m)} = \sum_{i=0}^{m} \alpha_i x^{(i)}$, which is better than $x^{(m)}$
- ❑ Let us write the error $y^{(m)}$

$$y^{(m)} - x^* = \sum_{i=0}^{m} \alpha_i x^{(i)} - x^* = \sum_{i=0}^{m} \alpha_i (x^{(i)} - x^*) = \sum_{i=0}^{m} \alpha_i G^i (x^{(0)} - x^*) = p_m(G)(x^{(0)} - x^*),$$

Here, $p_m(G) = \sum_{i=0}^{m} \alpha_i G^i$ is a polynomial in the matrix G, $p_m(1) = \sum_{i=0}^{m} \alpha_i = 1$

- ❑ $\rho(p_m(G)) \to \min$ – the spectral radius is minimized.
- ❑ $p_m(G)$ can be obtained using Chebyshev's polynomials $T_m(x)$

Chebyshev's acceleration (2)

- $p_m(G) = \mu_m T_m(G/\rho)$, where $\mu_m \equiv 1/T_m(1/\rho)$, $T_m(x)$ is a Chebyshev's polynomial, ρ is the spectral radius of the matrix G.
- Chebyshev's polynomials

$$T_0(x) = 1 \quad T_1(x) = x \quad T_m(x) = 2xT_{m-1}(x) - T_{m-2}(x)$$

- Three-term relation enables only three vectors $y^{(m)}$, $y^{(m-1)}$, $y^{(m-2)}$ to be used, but not all vectors $x^{(m)}$, $0 \le i \le m$.
- The following relations may be derived

$$y^{(m)} = \frac{2\mu_m}{\mu_{m-1}} \frac{G}{\rho} y^{(m-1)} - \frac{\mu_m}{\mu_{m-2}} y^{(m-2)} + \frac{2\mu_m}{\rho\mu_{m-1}} c \qquad \mu_m = \left(\frac{2}{\rho\mu_{m-1}} - \frac{1}{\mu_{m-2}} \right)^{-1}$$

- Requirements to G: $\lambda_i \in [-\rho, \rho]$
  - SOR is not applicable, but SSOR may be used

Chebyshev’s acceleration (3)

- Thus, Chebyshev’s method acceleration $x^{(s+1)} = Gx^{(s)} + c$ consists in:
  - Set $\mu_0 = 1, \mu_1 = \rho, y^{(0)} = x^{(0)}, y^{(1)} = Gx^{(0)} + c$.
  - Compute the following for *m*=2, 3, …

$$\mu_m = \left(\frac{2}{\rho\mu_{m-1}} - \frac{1}{\mu_{m-2}} \right)^{-1} \qquad y^{(m)} = \frac{2\mu_m}{\rho\mu_{m-1}}\left(Gy^{(m-1)} + c\right) - \frac{\mu_m}{\mu_{m-2}} y^{(m-2)}$$

- There is no need to explicitly compute *G* and *c*; iteration will have two stages

1) $\bar{y} = Gy^{(m-1)} + c$ 2) $y^{(m)} = \dfrac{2\mu_m}{\rho\mu_{m-1}}\bar{y} - \dfrac{\mu_m}{\mu_{m-2}} y^{(m-2)}$

SOFTWARE IMPLEMENTATION

Sequential version

Project creation (1)

- Run **Microsoft Visual Studio 2008**
- From the **File** menu, select **New→Project….**
- From the **New Project**, select **Win32** from the Project types pane and **Win32 Console Application** from the Templates pane; enter **BandOverRelaxation** in the **Solution** name field, enter **c:\ParallelCalculus\** (path to the folder with laboratory works). Press **OK.**
- From the **Win32 Application Wizard** dialog, press **Next** and click **Empty Project**. Press **Finish.**

Project creation (2)

- ☐ From the **Win32 Application Wizard** dialog, press **Next** (or select **Application Settings** in the tree on the left) and click **Empty Project**. Press **Finish.**
- ☐ From the **Solution Explorer**, execute **Add→New Item** in the **Source Files** folder. In the selection tree, select **Code**; select **C++ File (.cpp)** in the templates on the right, enter **main** in the **Name** field. Press **Add.**
- ☐ From the **Solution Explorer**, execute **Add→New Item** in the **Header Files** folder. In the selection tree, select **Code**; select **Header File (.h)** in the templates on the right, enter **SOR** in the **Name** field. Press **Add.** This file will contain prototypes of functions required for implementation of the Successive Over Relaxation method.

Project creation (3)

- Similarly, add the following files to the project:
- **SOR.c** that will contain implementation of functions required for the Successive Over Relaxation method.
- **SSOR.h** that will contain prototypes of functions required for method implementation
- SSOR.c that will contain implementation of functions indicated in the **SSOR.h** header file
- **ChebSSOR.h** to contain prototypes of functions required to implement the Symmetric Successive Over Relaxation method with Chebyshev’s acceleration

Project creation (4)

- **MatrixVectorOperations.h** that will contain prototypes of auxiliary functions to handle vectors and matrices
- **MatrixVectorOperations.c** that will contain implementations of auxiliary functions
- **mklMatrixOperations.h** where some functions from the MLK library will be used to form the matrix for the Poission equation.
- **mklMatrixOperations.c** that will store implementation of functions indicated in the **mklMatrixOperations.h** header file.

Project creation (5)

- **Norms.h** that will store prototypes of functions required to find norms of vectors
- **Norms.c** that will store implementation of functions indicated in the **mklMatrixOperations.h** header file.
- **Consts.h** that will store several invariables such as algorithm accuracy.

Connection to the Intel® Math Kernel Library (1)

- To form the linear system matrix, we shall use some of the MLK library functions.
- Library connection:
  - Open **Tools** → **Options** and select **Projects and Solutions→VC++ Directories**.
  - In the drop-down menu, first select **Include Files**, add a new entry containing the path to MLK library header files (e. g. C:\Program Files (x86)\Intel\ComposerXE-2011\mkl\include),

Connection to the Intel® Math Kernel Library (2)

- Then select **Library Files** and add the path to the library files:
  - To assemble a 32-bit application, enter the path to the static library for the ia-32 platform (e. g. **C:\Program Files (x86)\Intel\ComposerXE-2011\mkl\lib\ia32**)
  - To assemble a 64-bit application, enter the path to the static library for the 64-bit platform (e. g. **C:\Program Files (x86)\Intel\ComposerXE-2011\mkl\lib\intel64**).

Connection to the Intel® Math Kernel Library (3)

- From **Configuration Properties** in the **Linker→Input→Additional Dependencies tab**, enter the following static libraries:
  - for a 32-bit application, they are **mkl_core.lib, mkl_intel_c.lib, mkl_sequential.lib.**
  - for z 64-bit application, they are **mkl_core.lib, mkl_sequential.lib, mkl_intel_lp64.lib, mkl_blas95_lp64.lib.**

Elementary function (1)

```
void main(int argc, char* argv[])
{     // tested matrix file    .mtx
    char* fileName;
    //
    // found using one of the tested methods
    double excAccuracy;
    // computing functions performance time
    double time;
    // linear system dimension
    int size;
    // number of computing function iterations
    int stepCount;
    // maximum number of iterations
    int stepMax;
    // matrix file opening error
    int error;
  // continued in the following slide
```

Elementary function (2)

```
// matrices in CRS format
    crsMatrix crsM, crsMFull;
    // right-hand sides vector
    double* bVector;
    // linear system solution vector
    double* xVector;
    // omega relaxation parameter
    double omega;
    // successful algorithm run flag
    int flag;
    // single vector
    double* unitVector;
    // parameter for Chebyshev's accelerations
    double ro;
    // grid dimension for digitalization
    // Poisson equation
    int nMesh;
// continued in the following slide
```

Elementary function (3)

```
      // Poisson equation solution vector
     double* poissonVector;
     stepMax = 100000;
     if ((argc > 2) && (argc <= 6))
     {
        omega = atof(argv[4]);
        if (atoi(argv[1]) == 1)  // test matrices from
                                // .mtx file
{                 fileName = argv[3];
             crsM.Col=crsM.RowIndex=NULL;
             // read the matrix
             error = ReadMatrixFromFile(fileName,
                  &crsM.N,&crsM.Col,
                     &crsM.RowIndex, &crsM.Value, &crsM.NZ);
             // complete the matrix
             getFullMatrixFromUpperTriangular(&crsM,
                                         &crsMFull);
             size = crsM.N;
```

Elementary function (4)

```
bVector = (double*)malloc(size
                      *sizeof(double));
               xVector = (double*)malloc(size *
                       sizeof(double));
            unitVector = (double*)malloc(size *
                       sizeof(double));
            // form the single vector of exact solution
            UnitVector(xVector, size);
            UnitVector(unitVector, size);
            // compute the right-side vector of the linear system
            MultiplicateMV(&crsMFull, xVector, bVector);
            memset(xVector, 0, size * sizeof(double));
            stepCount = stepMax;
            if (atoi(argv[2]) == 1)
            {// call the function of solving linear systems by the SOR
   method
               time = SORSolve(&crsMFull, bVector,
                       xVector, stepMax, EPSILON, omega,
                       unitVector, &flag, &stepCount);
```

Elementary function (5)

```
// compute the difference between the obtained and
                // exact solution
                   excAccuracy = NormInfinityToUnit(xVector,
                              size);
                // print computation results
                printf("%s; %s; %d; %d; %e; %e; %.4f;
                     %d\n", fileName,        " ", size,
                     stepCount, time, excAccuracy, omega,
                     flag);
            }
            else if (atoi(argv[2]) == 2)
            {
                // call the function of solving linear systems by
            // Symmetric Successive Over Relaxation
                time = SymmetricSorSolve(&crsMFull,
                       bVector, xVector, stepMax, EPSILON,
                       omega, unitVector, &flag,
                       &stepCount);
```

Elementary function (6)

```
// compute the difference between the obtained and exact solution
                   excAccuracy = NormInfinityToUnit(xVector, size);
                // print computation results
                printf("%s; %s; %d; %d; %e; %e; %.4f;
                      %d\n", fileName, " ", size,
                      stepCount, time, excAccuracy, omega,
                      flag);
            }
            else if (atoi(argv[2]) == 3)
            {
            // ro parameters for Chebyshev's accelerations
                ro = atof(argv[5]);
                // call the function of solving linear systems by
            // Symmetric Successive Over Relaxation
                // with Chebyshev's acceleration
                time = ChebSSOR(&crsMFull, bVector,
                       xVector, ro,stepMax, EPSILON, omega,
                       unitVector, &flag, &stepCount);
```

Elementary function (7)

```
time = ChebSSOR(&crsMFull, bVector,
                       xVector, ro,stepMax, EPSILON, omega,
                       unitVector, &flag, &stepCount);
                // compute the difference between the obtained and
                // exact solution
                excAccuracy = NormInfinityToUnit(xVector,
                              crsMFull.N);
                // print computation results
                printf("%s; %s; %d; %d; %e; %e; %.4f; %d;
                      %f\n", fileName, " ", size,
                      stepCount, time, excAccuracy, omega,
                      flag, ro);
            }
               FreeMatrix(&crsM);
               FreeMatrix(&crsMFull);
               free(bVector);
               free(xVector);
       }
```

Elementary function (8)

```
        else // solve the Poisson equation
        {
                // obtain the partition number
             nMesh = atoi(argv[3]);
                // compute the problem dimension
             size = nMesh * nMesh;
             // form the upper triangle of the linear system matrix
             PoissonMatrix(&crsM, nMesh);
             // restore the complete linear system matrix by its
              // by its upper triangle
                getFullMatrixFromUpperTriangular(&crsM,
                                         &crsMFull);
             bVector = (double*)malloc(size *
                        sizeof(double));
                xVector = (double*)malloc(size *
                       sizeof(double));
             poissonVector = (double*)malloc(size *
                             sizeof(double));
```

Elementary function (9)

```
poissonVector = (double*)malloc(size *
                            sizeof(double));
            // compute the exact problem solution
            VectorV(nMesh, poissonVector);
            // based on the exact problem solution find the vector
            // of the right-hand sides
               MultiplicateMV(&crsMFull, poissonVector,
                          bVector);
            memset(xVector, 0, size * sizeof(double));
            // if the omega relaxation parameter is equal to zero
            // compute the best omega value
            if (omega == 0)
            {
                omega = PoissonOmega(nMesh);
            }
```

Elementary function (10)

```
    if (atoi(argv[2]) == 1)
                {
                    // call the function of solving linear systems by
                   // SOR method
                    time = SORSolve(&crsMFull, bVector,
                           xVector, stepMax, EPSILON,
                           omega, poissonVector, &flag,
                     &stepCount);
                           excAccuracy = NormInfinity(poissonVector,
                                  xVector, size);
              printf("%s; %s; %d; %d; %e; %e; %.4f;
                       %d\n", " ", " ",size, stepCount,
                     time, excAccuracy, omega, flag);
                }
```

Elementary function (11)

```
else if (atoi(argv[2]) == 2)
            {
                // call the function of solving linear systems by
                   // Symmetric Successive Over Relaxation
                time = SymmetricSorSolve(&crsMFull,
                       bVector, xVector, stepMax, EPSILON,
                              omega, poissonVector,&flag,
                              &stepCount);
                   excAccuracy = NormInfinity(poissonVector,
                                     xVector, size);
                printf("%s; %s; %d; %d; %e; %e; %.4f;
                      %d\n", " ", " ",      size, stepCount, time,
                      excAccuracy, omega, flag);
            }
```

Elementary function (12)

```
         else
            {
                ro = atof(argv[5]); /call the function of solving
linear systems by
                   // Symmetric Successive Over Relaxation
                   // with Chebyshev’s acceleration
                time = ChebSSOR(&crsMFull, bVector,
                       xVector, ro, stepMax, EPSILON,
                       omega, poissonVector,        &flag,
                       &stepCount);
                excAccuracy = NormInfinity(poissonVector,
                             xVector, size);
                printf("%s; %s; %d; %d; %e; %e; %.4f; %d;
                      %f\n", " ", " ", size, stepCount,
                      time, excAccuracy, omega, flag, ro);
            }  // free the memory
               FreeMatrix(&crsM); FreeMatrix(&crsMFull);
               free(bVector); free(xVector);
       }
    }
    return 0;
};
```

Auxiliary functions

- **Poisson.h** will store prototypes and **Poisson.c** will store implementation of functions defining the right part of the differential equation and forming the linear system matrix:

```
// compute the function u value
// u(x,y) = sin(PI * x) * sin(PI * y)
// in the determination region point (x, y)
double Function1(double x, double y)
{
        return sin(PI * x) * sin(PI * y);
}
```

Auxiliary functions. Exact vector for the Poisson equation linear system (1)

```
// the function computes the exact solution, the vector v for
// digitized linear system for the Poisson equation
// the vector v can be calculated exactly as
// we know the required function
// memory is considered to be allocated for the vector v
// this vector V is computed for a [0,1];[0,1] square
int VectorV(int N, double *v)
{
        double h;
        double x, y;
        int k;
        int i, j;
        h = (double)((double)1.0/(N + 1));
        x = 0;
        y = 0;
        k = 0;
        for(i = 1; i < N + 1; i++)
        {
                x = i * h;
                if ((i % 2) == 1) y = 0;
                else y = (N + 1) * h;
```

Auxiliary functions. Exact vector for the Poisson equation linear system (2)

```
for(j = 1; j < N + 1; j++)
               {
                      if ((i % 2) == 1)
            {
                y = y + h;
            }
                      else
            {
                y = y - h;
            }
                      v[k] = Function1(x, y);
                      k++;
               }
        }
        return 0;
}
```

Implementation of the Successive Over Relaxation method (1)

```
double SORSolve(crsMatrix* M, double* b, double* x, int
                iteration, double accuracy, double w,
                double* y, int* flag, int *iter)
{
int size;
    double* prevX;
    double Q;
    double diagElem;
    clock_t start, finish;
    int i, m, l;

    size = M->N;
    *flag = 0;
    *iter = iteration;
    prevX = (double*)malloc(size * sizeof(double));

    memset(x, 0, size * sizeof(double));
    memset(prevX, 0, size * sizeof(double));
```

Implementation of the Successive Over Relaxation method (2)

```
start = clock();
for(m = 0; m < iteration; m++)
{
         for(i = 0; i < size; i++)
         {
                 Q = 0;
                 for(l = M->RowIndex[i]; l < M->RowIndex[i
                          + 1]; l++)
                 {
                          if (M->Col[l] < i)
        {
                                   Q += M->Value[l] * x[M-
                                   >Col[l]];
        }
         else if (M->Col[l] > i)
        {
            Q += M->Value[l] * prevX[M->Col[l]];
        }
         else
        {
            diagElem = M->Value[l];
        }
```

Implementation of the Successive Over Relaxation method (3)

```
                      Q = (b[i] - Q) / diagElem;
                      x[i] = prevX[i] + w * (Q - prevX[i]);
               }
               memcpy(prevX, x, size * sizeof(double));

        if (NormInfinity(y, x, size) <= accuracy)
               {
                      *flag = 1;
                      *iter = m;
                      break;
               }
       }
       free(prevX);
       finish = clock();
    return (double)((finish - start) / CLOCKS_PER_SEC);
}
```

Implementation of the Symmetric Successive Over Relaxation method (1)

To solve linear systems using the SSOR method and perform mass experiments, implement the **SymmetricSORSolve()** function in **SSOR.c.**

```
double SymmetricSorSolve(crsMatrix* M,  double* b, double*
                       x, int iteration, double accuracy,
                       double w, double* y, int* flag, int*
                       iter)
{
          int size = M->N;
          double* prevX;
          double* semiX;
          double Q;
          double diagElem;
          int i, m, l;
          clock_t start, finish;

    prevX = (double*)malloc(size * sizeof(double));
    semiX = (double*)malloc(size * sizeof(double));


    memset(semiX, 0, size * sizeof(double));
    memset(prevX, 0, size * sizeof(double));
    start = clock();
    *flag = 0;
    *iter = iteration;
          for(m = 0; m < iteration; m++)
          {
```

Implementation of the Symmetric Successive Over Relaxation method (2)

First, the intermediate approximation is calculated based on the previous one and using the computation scheme of the SOR method. This part of the program can be called the first half iteration.

```
// First half iteration
                 for(i = 0; i < size; i++)
                 {
                          Q = 0;
                          for(l = M->RowIndex[i]; l < M->RowIndex[i
                                                + 1]; l++)
                          {
                                   if (M->Col[l] < i)
                {
                                           Q += M->Value[l] * semiX[M-
                                  >Col[l]];
                }
                                   else if (M->Col[l] > i)
                {
                    Q += M->Value[l] * prevX[M->Col[l]];
                }
                                   else
                {
```

Implementation of the Symmetric Successive Over Relaxation method (3)

First, the intermediate approximation is calculated based on the previous one and using the computation scheme of the SOR method. This part of the program can be called the first half iteration.

```
// First half iteration
                    for(i = 0; i < size; i++)
                    {
                              Q = 0;
                              for(l = M->RowIndex[i]; l < M->RowIndex[i
                                                + 1]; l++)
                              {
                                        if (M->Col[l] < i)
                {
                                        Q += M->Value[l] * semiX[M->Col[l]];
                                                                                        }
                                        else if (M->Col[l] > i)
                {
                    Q += M->Value[l] * prevX[M->Col[l]];
                }
                                        else
                {
                    diagElem = M->Value[l];
                }
                              }
                              Q = (b[i] - Q) / diagElem;
                              semiX[i] = prevX[i] + w * (Q - x[i]);
                    }
```

Implementation of the Symmetric Successive Over Relaxation method (4)

Then, after the first half iteration, the next approximation is computed on the basis of the intermediate approximation. To compute the next approximation, use the computation scheme with a reverse order of approximation vector components.

```
          // Second half iteration
for(i = size - 1; i >= 0; i--)
{
          Q = 0;
          for(l = M->RowIndex[i]; l < M->RowIndex[i+1]; l++)
                    {
          if (M->Col[l] < i)
      {
          Q += M->Value[l] * semiX[M-
                         >Col[l]];
      }
          else if (M->Col[l] > i)
      {
          Q += M->Value[l] * x[M->Col[l]];
      }
          else
      {
          diagElem = M->Value[l];
      }
}
          Q = (b[i] - Q) / diagElem;
          x[i] = semiX[i] + w * (Q - semiX[i]);
}
if (NormInfinity(y, x, size) <= accuracy)
```

Implementation of the Symmetric Successive Over Relaxation method (5)

```
if (NormInfinity(y, x, size) <= accuracy)
        {
            *iter = m;
            *flag = 1;
            break;
        }
                 memcpy(prevX, x, size * sizeof(double));
         }
         free(prevX);
         free(semiX);
         finish = clock();
return (double)((finish - start)/ CLOCKS_PER_SEC);
```

Software implementation of the Symmetric Successive Over Relaxation method with Chebyshev’s acceleration

Place prototypes of the respective functions in **ChebSSOR.h** and their implementation - in **ChebSSOR.c.** To solve linear systems using the SOR method and perform mass experiments, implement the ChebSSOR **()** function in **ChebSSOR.c.**

```
double ChebSSOR(crsMatrix* M, double* b, double* x,
                double ro, int iteration, double accuracy,
                double w, double* y, int* flag, int* iter)
{
          long double mu0, mu1, mu2;
          double k1, k2, k3;
          int size = M->N;
          double* y0, * y1, * y12, * y2, * tmp;

      int i, j;

      clock_t start, finish;

      double time;
          mu0 = 1;
          mu1 = ro;

      y0 = (double*) malloc(size * sizeof(double));
          y1  = (double*)malloc(size * sizeof(double));
          y12 = (double*)malloc(size * sizeof(double));
          y2  = (double*)malloc(size * sizeof(double));
          tmp = (double*)malloc(size * sizeof(double));

   memset(y0, 0, size * sizeof(double));
```

Software implementation of the Symmetric Successive Over Relaxation method with Chebyshev’s acceleration(2)

```
SymmetricSorIterate(M, b, y0, y1, w);
*flag = 0;
 start = clock();
for(i = 1; i < iteration; i++)
{         // Three-term recurrence for
          // mu0, mu1, mu2
          mu2 = 1.0 / (2.0/(ro * mu1) - 1.0/mu0);
          k1 = 2 *  mu2 / (ro * mu1);
          k2 = mu2 / mu0;
          mu0 = mu1;
          mu1 = mu2;
          // Three-term recurrence for y0,
          //y1, y2
          SymmetricSorIterate(M, b, y1, y12, w);
          for(j = 0; j < size; j++)
                    y2[j] = k1 * y12[j] - k2 * y0[j];
          if (NormInfinity(y, y2, size) <= accuracy)
          {
                    *iter = i;
                    *flag = 1;
                    break;
          }
          tmp = y1;
          y0 = y1;
          y1 = y2;
          y2 = tmp;
}
```

Software implementation of the Symmetric Successive Over Relaxation method with Chebyshev’s acceleration(3)

```
finish = clock();
      time = (double)((finish - start) / CLOCKS_PER_SEC);
        memcpy(x, y2, size * sizeof(double));
        free(y0);
        free(y1);
        free(y12);
        free(y2);
        free(tmp);
        return time;
}
```

Project compilation and application run

- Add missing functions to the software implementation; include necessary header files.
- Having developed the software implementation, build the project by executing **Build→Rebuild SOR** and check the application for consistent running.

Method convergence analysis (1)

- To analyze method convergence, first use matrices from The University of Florida Sparse Matrix Collection:

  http://www.cise.ufl.edu/research/sparse/matrices/

- All matrices are symmetric positive definite.
- The best ω and ρ values are difficult to compute analitically for a general matrix, so they were determined by expertiment.
- Method accuracy $\varepsilon=10^{-6}$
- Convergence-related results are listed in the tables below

Method convergence analysis (2)

Convergence of the Successive Over Relaxation method, required accuracy 0,000001

Matrix name	Dimension	Condition number	Number of algorithm steps	Allowed difference from the exact solution	Omega relaxation parameter
LFAT5.mtx	14	1.4e+08	166	8.021907e-007	1.9
mesh1em6.mtx	48	6.1	146	9.922382e-007	1.9
bcsstk04.mtx	132	5.6e+06	341	9.461812e-007	1.9
bcsstk05.mtx	153	3.5e+04	986	9.961649e-007	1.87
bcsstk06.mtx	420	1.2e+07	3383	9.974012e-007	1.926
bcsstk09.mtx	1083	3.1e+04	885	9.937784e-007	1.95
chem97ZtZ.mtx	2541	2.5e+2	144	9.068114e-007	1.9

Method convergence analysis (3)

Convergence of the Symmetric Successive Over Relaxation method, required accuracy 0,000001

Matrix name	Dimension	Condition number	Number of algorithm steps	Allowed difference from the exact solution	Omega relaxation parameter
LFAT5.mtx	14	1.4e+08	4263	9.991575e-007	1.9
mesh1em6.mtx	48	6.1	140	9.936279e-007	1.9
bcsstk04.mtx	132	5.6e+06	18211	9.994442e-007	1.8
bcsstk05.mtx	153	3.5e+04	13731	9.992477e-007	1.8
bcsstk06.mtx	420	1.2e+07	43181	9.996227e-007	1.32
bcsstk09.mtx	1083	3.1e+04	30761	9.996047e-007	1.1
chem97ZtZ.mtx	2541	2.5e+2	323	9.996019e-007	1.9

Method convergence analysis (4)

Convergence of the Symmetric Successive Over Relaxation method with Chebyshev's acceleration, required accuracy 0,000001

Matrix name	Dimension	Condition number	Number of algorithm steps	Allowed difference from the exact solution	Relaxation parameter	Parameter
LFAT5.mtx	14	1.4e+08	89	6.112398e-007	1.5	0.9779
mesh1em6.mtx	48	6.1	35	7.905490e-007	1.9	0.9
bcsstk04.mtx	132	5.6e+06	229	9.546005e-007	1.092	0.996832
bcsstk05.mtx	153	3.5e+04	251	9.499373e-007	1.11	0.998188
bcsstk06.mtx	420	1.2e+07	730	8.346634e-007	1.11	0.999666
bcsstk09.mtx	1083	3.1e+04	537	9.961580e-007	1.11	0.999588
chem97ZtZ.mtx	2541	2.5e+2	125	9.711613e-007	1.9	0.9

Method convergence analysis (5)

Convergence rate of the Successive Over Relaxation method as compared to that of the Symmetric Successive Over Relaxation method with Chebyshev's acceleration

Matrix name	SOR		SOR-Cheb		
	ω	*Number of algorithm steps*	ω	ρ	*Number of algorithm steps*
LFAT5.mtx	1.9	166	1.5	0.9779	89
mesh1em6.mtx	1.9	146	1.9	0.9	35
bcsstk04.mtx	1.9	341	1.092	0.996832	229
bcsstk05.mtx	1.87	986	1.11	0.998188	251
bcsstk06.mtx	1.926	3383	1.11	0.999666	730
bcsstk09.mtx	1.95	885	1.11	0.999588	537
chem97ZtZ.mtx	1.9	144	1.9	0.9	125

Method convergence analysis (6)

Some conclusions:

❑All algorithms are convergent and ensure required accuracy

❑The convergence rate will depend on the matrix dimension and condition number.

❑In the general case, one can determine an approximate valued of the best relaxation parameter for a specific matrix by experiment

❑The Symmetric Successive Over Relaxation algorithm demonstrates a lower convergence rate than that of the Successive Over Relaxation algorithm

❑ If the procedure of Chebyshev’s accelerations is followed, the required number of iterations is reduced (from 1.5 to 3.5 times)

❑Parameter selection has a considerable effect on the method convergence. Exact estimation of the iteration matrix spectral radius considerably reduces the number of iterations to ensure the required accuracy.

Method convergence analysis (7)
Poisson equation

- The linear system results from PDE discretization.
- The system solution is a function that is known in advance:

$$u(x, y) = \sin(\pi x) * \sin(\pi y)$$

- Given a function known in advance, the exact linear system solution vector is known
- for this type of problems it is known that

$$\omega_{opt} = \frac{2}{1 + 2\sin(\pi h/2)}$$

- for this type of problems it is known that

$$\rho_{opt} = 1 - \frac{\pi h}{2}$$

- the linear system matrix has a five-diagonal pattern

Method convergence analysis (8)
Poisson equation

- ❑ Method parameters: $\rho=0.99$, $\varepsilon=10^{-6}$.

n	nz/n	ω	s		
			SOR	SSOR	SSOR-Cheb
10000	$4{,}9\cdot10^{-6}$	1.9397	286	342	53
22500	$9{,}8\cdot10^{-6}$	1.9592	428	512	65
40000	$3{,}1\cdot10^{-7}$	1.9692	569	682	72
62500	$1{,}2\cdot10^{-7}$	1.9753	711	852	123
90000	$6{,}1\cdot10^{-8}$	1.9793	853	1022	91
122500	$3{,}3\cdot10^{-8}$	1.9823	995	1192	85
160000	$1{,}9\cdot10^{-8}$	1.9845	1137	1362	97
202500	$1{,}2\cdot10^{-8}$	1.9862	1278	1532	143
250000	$7{,}9\cdot10^{-9}$	1.9875	1420	1702	276

Test questions

- Why is it impossible to apply the procedure of Chebyshev's accelerations directly to the Successive Over Relaxation method?
- Deduce a linear system resulting from grid approximation of the heat transfer equation. What structural peculiarities does this matrix have?
- Give the canonical SOR method form and approximation component computation formula.
- Which sparse matrix storage formats do you know? When is each of them used?

Added tasks

- ❑ Compute by experiment the best relaxation parameter value for a general matrix. What happens to the convergence rate if the parameter value is close to the best one?
- ❑ Compute the exact spectral radius of the iteration matrix for a system of linear equations resulting from the Poisson equation. MatLab. may be used. How does accurate parameter selection influence the method convergence?
- ❑ For a test matrix, observe dependence of the iteration matrix spectral radius on the relaxation parameter. Perform a series of experiments, record the results in a table and see what happens to the spectral radius in the course of approximation to the best value.

Questions

❑ ???