# Introduction to inverse and ill-posed problems
# Project “Solution of time-harmonic coefficient inverse problem” in C++/PETSc

# Project “Solution of time-harmonic coefficient inverse problem” in C++/PETSc

- PETSc libraries are a suite of data structures and routines for the scalable (parallel) solution of scientific applications.
- Link to the PETSc documentation:

  `http://www.mcs.anl.gov/petsc/documentation/`

# Example: solution of Poisson's equation [BKK]

The model problem is the following Dirichlet problem for Poisson's equation:

$$\begin{aligned} -\triangle u(x) &= f(x) \ \text{ in } \ \Omega, \\ u &= 0 \text{ on } \partial\Omega. \end{aligned} \tag{1}$$

Here $f(x)$ is a given function, $u(x)$ is the unknown function, and the domain $\Omega$ is the unit square $\Omega = \{(x_1, x_2) \in (0,1) \times (0,1)\}$. To solve numerically (1) we first discretize the domain $\Omega$ with $x_{1\,i} = ih_1$ and $x_{2j} = jh_2$, where $h_1 = 1/(n_i - 1)$ and $h_2 = 1/(n_j - 1)$ are the mesh sizes in the directions $x_1, x_2$, respectively, $n_i$ and $n_j$ are the numbers of discretization points in the directions $x_1, x_2$, respectively. In this example we choose $n_i = n_j = n$ with $n = N + 2$, where $N$ is the number of inner nodes in the directions $x_1, x_2$, respectively.
Indices $(i, j)$ are such that $0 < i, j \leq n$ and are associated with every global node $n_{glob}$ of the finite difference mesh. Global nodes numbers $n_{glob}$ in two-dimensional case can be computed as:

$$n_{glob} = j + n_i(i - 1). \tag{2}$$

We use the standard finite difference discretization of the Laplace operator $\Delta u$ in two dimensions and obtain discrete laplacian $\Delta u_{i,j}$:

$$\Delta u_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \tag{3}$$

where $u_{i,j}$ is the solution at the discrete point $(i, j)$. Using (3), we obtain the following scheme for solving problem (1):

$$-\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}\right) = f_{i,j}, \tag{4}$$

where $f_{i,j}$ are the value of the function $f$ at the discrete point $(i, j)$.
Then (4) can be rewritten as

$$-\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j} + u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) = h^2 f_{i,j}, \tag{5}$$

or in the more convenient form as

$$-u_{i+1,j} + 4u_{i,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = h^2 f_{i,j}. \tag{6}$$

System (6) can be written in the form $Au = b$. The vector $b$ has the components $b_{i,j} = h^2 f_{i,j}$. The explicit elements of the matrix $A$ are given by the following block matrix

$$A = \left(\begin{array}{c|c|c} A_N & -I_N & \\ \hline -I_N & \ddots & \ddots \\ & \ddots & \ddots & -I_N \\ \hline & & -I_N & A_N \end{array}\right)$$

with blocks $A_N$ of order $N$ given by

$$A_N = \begin{pmatrix} 4 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 4 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 4 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & \cdots & 0 & -1 & 4 \end{pmatrix},$$

which are located on the diagonal of the matrix $A$, and blocks with the identity matrices $-I_N$ of order $N$ on its off-diagonals. The matrix $A$ is symmetric and positive definite and we can use the $LU$ factorization algorithm without pivoting.

Suppose, that we have discretized the two-dimensional domain $\Omega$ as described above with $N = n_i = n_j = 3$. We present the schematic discretization via the global nodes numbering

$$n_{glob} = j + n_i \cdot (i - 1)$$

in the following scheme:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \Longrightarrow \begin{pmatrix} n_1 & n_2 & n_3 \\ n_4 & n_5 & n_6 \\ n_7 & n_8 & n_9 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}. \quad (7)$$

Then the explicit form of the block matrix $A$ will be:

$$A = \left(\begin{array}{ccc|ccc|ccc} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{array}\right).$$

# Example 8.2: Gaussian elimination for solution of Poisson's equation

We illustrate the numerical solution of problem (1). We define the right hand side $f(x)$ of (1) as

$$f(x_1, x_2) = A_f \exp\left(-\frac{(x_1 - c_1)^2}{2s_1^2} - \frac{(x_2 - c_2)^2}{2s_2^2}\right)\frac{1}{a(x_1, x_2)}, \tag{8}$$

The coefficient $a(x_1, x_2)$ in (8) is given by the following Gaussian function:

$$a(x_1, x_2) = 1 + A \exp\left(-\frac{(x_1 - c_1)^2}{2s_1^2} - \frac{(x_2 - c_2)^2}{2s_2^2}\right), \tag{9}$$

Here $A$, $A_f$ are the amplitudes of these functions, $c_1, c_2$ are constants which show the location of the center of the Gaussian functions, and $s_1, s_2$ are constants which show spreading of the functions in $x_1$ and $x_2$ directions.
We produce the mesh with the points $(x_{1\,i}, x_{2j})$ such that $x_{1\,i} = ih, x_{2j} = jh$ with $h = 1/(N+1)$, where $N$ is the number of the inner points in $x_1$ and $x_2$ directions. The linear system of equations $Au = f$ is solved then via the LU factorization of the matrix $A$ without pivoting.

# Example 8.2: solution of Poisson's equation using LU factorization

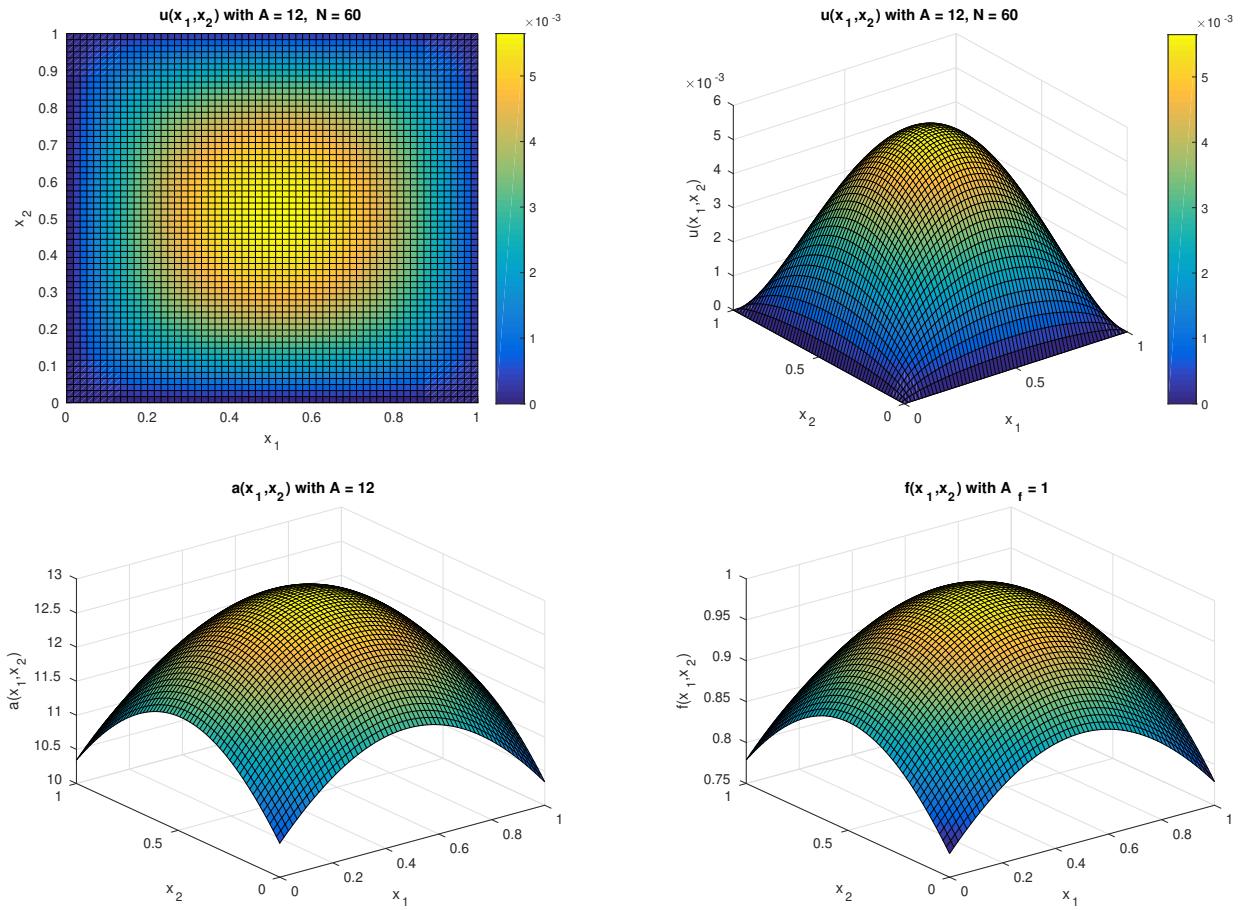



Figure: Solution of Poisson's equation (1) with $f(x_1, x_2)$ as in (8) and $a(x_1, x_2)$ as in (9).

# Example 8.4.4: solution of Poisson's equation using Cholesky factorization

$$f(x_1, x_2) = 1 + 10e^{\left(-\frac{(x_1-0.25)^2}{0.02} - \frac{(x_2-0.25)^2}{0.02}\right)} + 10e^{\left(-\frac{(x_1-0.75)^2}{0.02} - \frac{(x_2-0.75)^2}{0.02}\right)} \quad (10)$$

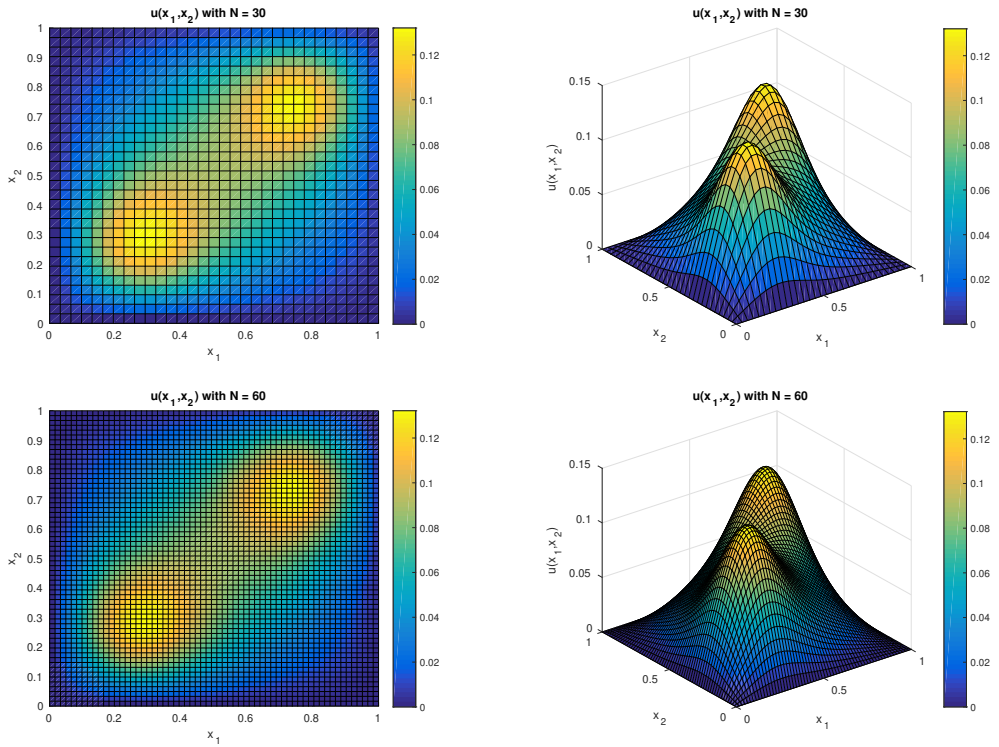



Figure: Solution of Poisson's equation (1) with $f(x_1, x_2)$ as in (10).

# Solution of the test problem

Now we illustrate how C++/PETSc solver can be used for solution of the following Dirichlet problem for Helmholtz equation in two dimensions:

$$\begin{aligned} \triangle u(x) + \omega^2 \varepsilon(x) u = f(x) \ \text{ in } \ \Omega, \\ u = 0 \text{ on } \partial\Omega. \end{aligned} \tag{11}$$

Here $f(x)$ is a given function, $u(x)$ is the unknown function to be computed, and the domain $\Omega$ is the unit square
$\Omega = \{(x_1, x_2) \in (0,1) \times (0,1)\}$.
The exact solution of (11) with the right hand side

$$\begin{aligned} f(x_1, x_2) = -(8\pi^2)\sin(2\pi x_1)\sin(2\pi x_2) - 2ix_1(1-x_1) - 2ix_2(1-x_2) \\ + \omega^2 \varepsilon(x)(\sin(2\pi x_1)\sin(2\pi x_2) + ix_1(1-x_1)x_2(1-x_2)) \end{aligned} \tag{12}$$

is the function

$$u(x_1, x_2) = \sin(2\pi x_1)\sin(2\pi x_2) + ix_1(1-x_1)x_2(1-x_2). \tag{13}$$

# Description of C++/PETSc solver

We set the computational domain to be the unit square $\Omega = \{(x_1, x_2) \in (0,1) \times (0,1)\}$ and discretize it as it described in the previous section. The main program

```
cplxmaxwell.cpp
```

is compiled using version of PETSc

```
petsc-3.10.4c
```

on 64 bits Red Hat Linux Workstation as

```
make runmaxwell
```

# Makefile

An example of Makefile used for compilation of PETSc program cplxmaxwell.cpp which we present below is:

```
  PETSC_ARCH=/chalmers/sw/sup64/petsc-3.10.4c
include ${PETSC_ARCH}/lib/petsc/conf/variables
include ${PETSC_ARCH}/lib/petsc/conf/rules
MPI_INCLUDE = ${PETSC_ARCH}/include/mpiuni
CXX = g++
CXXFLAGS = -Wall -Wextra -g -O0 -c
-Iinclude -I${PETSC_ARCH}/include -I${MPI_INCLUDE}
LD = g++
LFLAGS =
OBJECTS = cplxmaxwell.o
RUNMAXWELL = runmaxwell
all: $(RUNMAXWELL)
%.o: %.cpp
$(CXX) $(CXXFLAGS) -o $@ $<
$(RUNMAXWELL): $(OBJECTS)
$(LD) $(LFLAGS) $(OBJECTS) $(PETSC_LIB) -o $@
```

For solution of system of linear equations $Ax = b$ was used inbuilt PETSc function with the scalable linear equations solvers (KSP) component. This component provides interface to the combination of a Krylov subspace iterative method and a preconditioner which can be chosen by user [PETSc]. It is possible choose between three different preconditioners which are encoded by numbers:

```
1-  Jacobi's method
2 - Gauss-Seidel method
3 - Successive Overrelaxation method (SOR)
```

To run the main program `cplxmaxwell.cpp` one need to write:

```
>runmaxwellv2 argv[1] argv[2]
```

Here, arguments are defined as follows:

```
argv[1] - preconditioner (should be 1,2 or 3)
argv[2] - number of discretization points in x and y directions
```

[PETSc] Portable, Extensible Toolkit for Scientific Computation PETSc at http://www.mcs.anl.gov/petsc/

# Preconditioning for Linear Systems

Preconditioning technique is used for the reduction of the condition number of the considered problem. For the solution of linear system of equations $Ax = b$ the preconditioner matrix $P$ of a matrix $A$ is a matrix $P^{-1}A$ such that $P^{-1}A$ has a smaller condition number then the original matrix $A$. This means that instead of the solution of a system $Ax = b$ we will consider solution of the system

$$P^{-1}Ax = P^{-1}b. \tag{14}$$

The matrix $P$ should have the following properties:

- $P$ is s.p.d. matrix;
- $P^{-1}A$ is well conditioned;
- The system $Px = b$ should be easy solvable.

The preconditioned conjugate gradient method is derived as follows. First we multiply both sides of (14) by $P^{1/2}$ to get

$$(P^{-1/2}AP^{-1/2})(P^{1/2}x) = P^{-1/2}b. \tag{15}$$

# Preconditioning for Linear Systems

We note that the system (15) is s.p.d. since we have chosen the matrix $P$ such that $P = QQ^T$ which is the eigendecomposition of $P$. Then the matrix $P^{1/2}$ will be s.p.d. if it is defined as

$$P^{1/2} = Q^{1/2}Q^T.$$

Defining

$$\tilde{A} := P^{-1/2}AP^{-1/2}, \tilde{x} := P^{1/2}x, \tilde{b} = P^{-1/2}b$$

we can rewrite (15) as the system $\tilde{A}\tilde{x} = \tilde{b}$. Matrices $\tilde{A}$ and $P^{-1}A$ are similar since $P^{-1}A = P^{-1/2}\tilde{A}P^{1/2}$. Thus, $\tilde{A}$ and $P^{-1}A$ have the same eigenvalues. Thus, instead of the solution of $P^{-1}Ax = P^{-1}b$ we will present preconditioned conjugate gradient (PCG) algorithm for the solution of $\tilde{A}\tilde{x} = \tilde{b}$.

# Preconditioned conjugate gradient algorithm

Initialization: $r = 0; \quad x_0 = 0; \quad R_0 = b; \quad p_1 = P^{-1}b; y_0 = P^{-1}R_0$

*repeat*

$r = r + 1$

$z = A \cdot p_r$

$\nu_r = (y_{r-1}^T R_{r-1})/(p_r^T z)$

$x_r = x_{r-1} + \nu_r p_r$

$R_r = R_{r-1} - \nu_r z$

$y_r = P^{-1} R_r$

$\mu_{r+1} = (y_r^T R_r)/(y_{r-1}^T R_{r-1})$

$p_{r+1} = y_r + \mu_{r+1} p_r$

*until* $\|R_r\|_2$ *is small enough*

# Common preconditioners

Common preconditioner matrices $P$ are:

- Jacobi preconditioner $P = (a_{11}, ..., a_{nn})$. Such choice of the preconditioner reduces the condition number of $P^{-1}A$ around factor $n$ of its minimal value.
- block Jacobi preconditioner

$$P = \begin{pmatrix} P_{1,1} & ... & 0 \\ ... & ... & ... \\ 0 & ... & P_{r,r} \end{pmatrix} \tag{16}$$

with $P_{i,i} = A_{i,i}$, $i = 1, ..., r$, for the block matrix $A$ given by

$$A = \begin{pmatrix} A_{1,1} & ... & A_{1,r} \\ ... & ... & ... \\ A_{r,1} & ... & A_{r,r} \end{pmatrix} \tag{17}$$

with square blocks $A_{i,i}$, $i = 1, ..., r$. Such choice of preconditioner $P$ minimizes the condition number of $P^{-1/2}AP^{-1/2}$ within a factor of $r$.

- Method of SSOR can be used as a block preconditioner as well. If the original matrix $A$ can be split into diagonal, lower and upper triangular as $A = D + L + L^T$Ã¯Â¿Â½ then the SSOR preconditioner matrix is defined as

$$P = (D + L)D^{-1}(D + L)^T$$

It can also be parametrised by $\omega$Ã¯Â¿Â½ as follows:

$$P(\omega) = \frac{\omega}{2-\omega}\left(\frac{1}{\omega}D + L\right)D^{-1}\left(\frac{1}{\omega}D + L\right)^T$$

- Incomplete Cholesky factorization with $A = LL^T$ is often used for PCG algorithm. In this case a sparse lower triangular matrix $\tilde{L}$ is chosen to be close to $L$. Then the preconditioner is defined as $P = \tilde{L}\tilde{L}^T$.
- Incomplete LU preconditioner.

# Solution of the problem (11) using the C++/PETSc program cplmaxwell.cpp via SOR with $n_x = n_y = 21$.

For example, to execute the main program `cplxmaxwell.cpp` using SOR method and 21 discretization points in $x$ and $y$ directions, one should run this program, as follows:

```
>runmaxwell 3 21
```

The results will be printed in the files

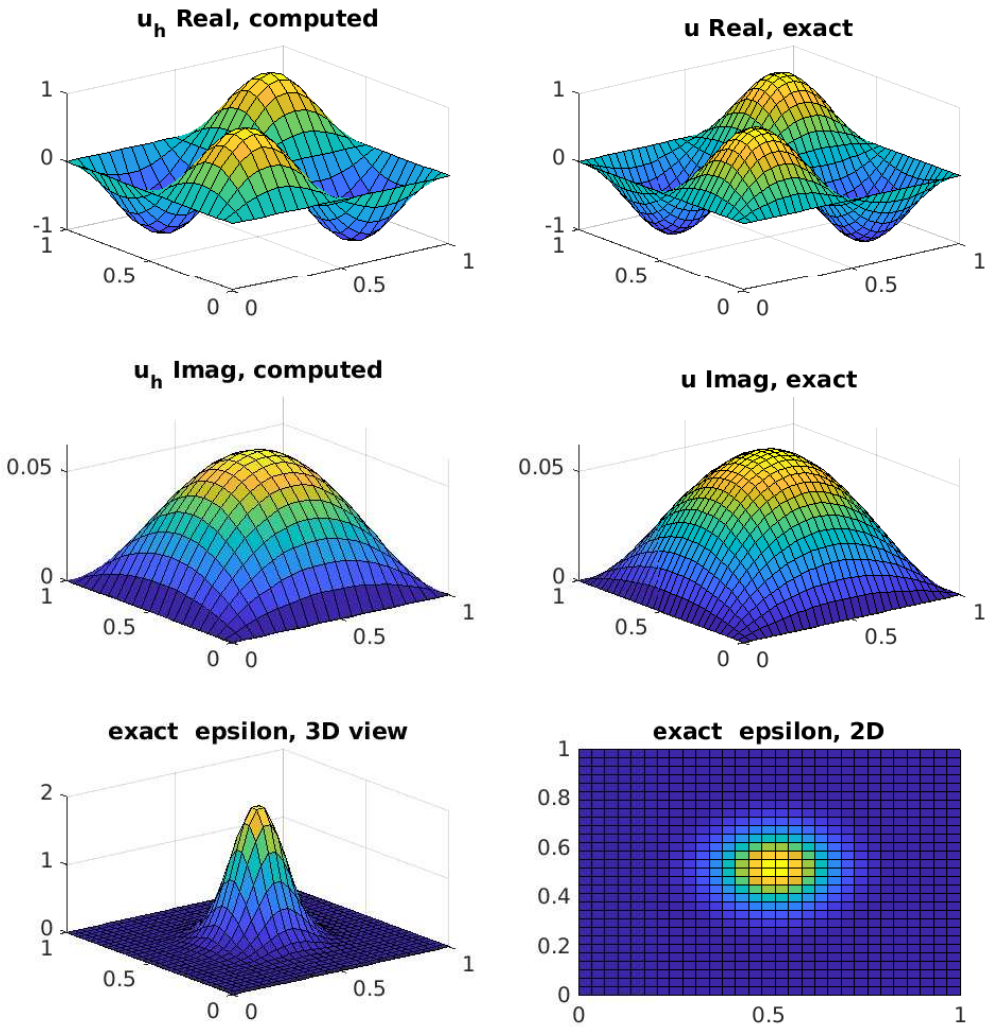
`nodes.m`
`values.m`

and can be visualized in Matlab using the file

`viewer.m`

which is available for download on the course homepage, see also below.

# Solution of the problem (11) using the C++/PETSc program cplmaxwell.cpp via SOR with $n_x = n_y = 21$.



# Program cplxmaxwell.cpp

```
// to run
// runmaxwell argv[1] argv[2]
// Arguments:
// argv[1] - preconditioner (should be 1,2 or 3)
// argv[2] - number of discretization points in x and y directions

static char help[] ="";
#include<iostream>
#include<fstream>
#include<petsc.h>
#include<petscvec.h>
#include<petscmat.h>
#include<petscksp.h>
#include<complex>

using namespace std;

char METHOD_NAMES[8][70] = {
    "invalid method",
    "Jacobi's method",
    "Gauss-Seidel method",
    "Successive Overrelaxation method (SOR)"};

char *GetMethodName(PetscInt method) {
    if (method < 0 || method > 3)
        return METHOD_NAMES[0];
    else
        return METHOD_NAMES[method];
}
```

```
PetscScalar   epsilon(const PetscReal x, const PetscReal y)
{
  PetscReal rpart, ipart;

PetscReal x_0=0.5;
PetscReal y_0=0.5;
PetscReal c_x=1;
PetscReal c_y=1;
 rpart=2*exp(-((x-x_0)*(x-x_0)/(2*c_x*c_x) +(y-y_0)*(y-y_0)/(2*c_y*c_y)));
  ipart = 0;
  PetscScalar scalareps(rpart, ipart);
  return scalareps;
```

```
PetscScalar right_hand_side(const PetscReal x, const PetscReal y,
    const PetscReal omega)
{
  PetscReal rpart, ipart, pi = 3.14159265359;


  PetscReal x_0=0.5;
  PetscReal y_0=0.5;
  PetscReal c_x=1;
  PetscReal c_y=1;
  PetscReal epsilon_real =
 2*exp(-((x-x_0)*(x-x_0)/(2*c_x*c_x) +(y-y_0)*(y-y_0)/(2*c_y*c_y)));


  rpart =  -(8*pi*pi)*sin(2*pi*x)*sin(2*pi*y)
+ omega*omega*epsilon_real*(sin(2*pi*x)*sin(2*pi*y));

  ipart = -2*(x - x*x + y - y*y)
 + omega*omega*epsilon_real*x*(1-x)*y*(1-y);

  PetscScalar f(rpart, ipart);
  return f;
```

```
PetscScalar wave_number(const PetscReal kreal, const PetscReal kimag)
{
  //PetscReal rpart, ipart;
    //rpart = 1;
    //ipart = 1;
    PetscScalar k(kreal, kimag);
    return k;
}
```

```
int main(int argc, char **argv)
{
  PetscErrorCode ierr;

  cout << "Initializing ..." << endl;
  // PetscInitialize(&argc, &argv, NULL, NULL);

 ierr = PetscInitialize(&argc, &argv,(char *)0, help);CHKERRQ(ierr);

  PetscInt method =  atoi(argv[1]);
  PetscBool methodSet = PETSC_FALSE;

ierr = PetscOptionsGetInt(NULL, NULL, "-m", &method, &methodSet);
    if (method < 1 || method > 7) {
        cout << "Invalid number of the selected method: "
     << method << ".\nExiting..." << endl;
        exit(-1);
    }

        PetscPrintf(PETSC_COMM_WORLD, "Using %s\n", GetMethodName(method));

  cout << "Setting parameters..." << endl;
   Vec b, u;
  Mat A;
  KSP ksp;
  PC preconditioner;
  PetscInt Nx = atoi(argv[2]), Ny = Nx, Nsys, node_idx = 0, col[5], nadj;

  Nsys = Nx*Ny; // dimension of linear system = number of nodes
  PetscReal x[Nx], y[Ny], nodes[Nsys][2];
  PetscScalar value, value_epsilon, diffpoints[5], h;
```

```
// Set up vectors
cout << "Setting up vectors..." << endl;
ierr = VecCreate(PETSC_COMM_WORLD, &b); CHKERRQ(ierr);
ierr = VecSetSizes(b, PETSC_DECIDE, Nsys); CHKERRQ(ierr);
ierr = VecSetType(b, VECSTANDARD); CHKERRQ(ierr);
ierr = VecDuplicate(b, &u);
```

```
// Set up matrix
cout << "Setting up matrix..." << endl;
ierr = MatCreate(PETSC_COMM_WORLD, &A); CHKERRQ(ierr);
ierr = MatSetSizes(A,PETSC_DECIDE, PETSC_DECIDE, Nsys, Nsys);
 CHKERRQ(ierr);
ierr = MatSetFromOptions(A); CHKERRQ(ierr);
ierr = MatSetUp(A); CHKERRQ(ierr);

// Create grid
cout << "Constructing grid..." << endl;
h = 1.0/(Nx - 1);
for (int i = 0; i < Nx; i++)
  x[i] = 1.0*i/(Nx - 1);
for (int j = 0; j < Ny; j++)
  y[j] = 1.0*j/(Ny - 1);
```

```
  // Assemble linear system ...
  cout << "Assembling system..." << endl;

  PetscScalar k;
  double omegareal=40;
  for (int i = 0; i < Nx; i++)
    {
      for (int j = 0; j < Ny; j++)
{
  nodes[node_idx][0] = x[i];
  nodes[node_idx][1] = y[j];

   k = omegareal*omegareal*epsilon(x[i], y[j]);
   value_epsilon =  h*h*k;

diffpoints[0] =  -4.0 + h*h*k;
        diffpoints[1] = 1.0;
        diffpoints[2] = 1.0;
        diffpoints[3] = 1.0;
        diffpoints[4] = 1.0;

  if (i > 0 && i < Nx - 1 && j > 0 && j < Ny - 1) // interior
    {
            col[0] = node_idx;
            col[1] = node_idx - 1;
            col[2] = node_idx + 1;
            col[3] = node_idx - Ny;
            col[4] = node_idx + Ny;

            nadj = 5;
            value = h*h*right_hand_side(x[i], y[j],omegareal);


    } else
```

```
// on boundary
    {
      col[0] = node_idx;
      nadj   = 1;
      value  = 0.0;
    }
ierr = MatSetValues(A, 1, &node_idx, nadj, col, diffpoints, INSERT_VALUES);
CHKERRQ(ierr);
ierr = VecSetValues(b, 1, &node_idx, &value, INSERT_VALUES);
CHKERRQ(ierr);

  node_idx++;
}
    }
  ierr = MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

  // Solve linear system
  cout << "Solving linear system ..." << endl;
  ierr = KSPCreate(PETSC_COMM_WORLD, &ksp); CHKERRQ(ierr);
  ierr = KSPSetOperators(ksp, A, A); CHKERRQ(ierr);
```

```
  // set preconditioner
   ierr = KSPGetPC(ksp, &preconditioner); CHKERRQ(ierr);

    if (method == 1)
      {
 ierr = PCSetType(preconditioner,PCJACOBI); CHKERRQ(ierr);
      }
    else if (method == 2)
      {

 ierr = PCSetType(preconditioner, PCSOR);
  CHKERRQ(ierr);
      }
else if (method == 3)
  {
    const PetscReal omega = 1.5;
    ierr = PCSetType(preconditioner, PCSOR); CHKERRQ(ierr);
    ierr = PCSORSetOmega(preconditioner, omega); CHKERRQ(ierr);
  }


  ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
  ierr = KSPSolve(ksp, b, u); CHKERRQ(ierr);
```

```
  // Print to files
  cout << "Writing to files..." << endl;
  FILE* nodefile = fopen("nodes.m", "w");
  for (int idx = 0; idx < Nsys; idx++)
    fprintf(nodefile, "%f \t %f \n", nodes[idx][0], nodes[idx][1]);
  fclose(nodefile);
  FILE* solfile = fopen("values.m", "w");
  for (int idx = 0; idx < Nsys; idx++)
    {
      ierr = VecGetValues(u, 1, &idx, &value);
      fprintf(solfile, "%f \t %f \n", real(value), imag(value));
    }
  fclose(solfile);

  // Clean up
  ierr = VecDestroy(&b); CHKERRQ(ierr);
  ierr = VecDestroy(&u); CHKERRQ(ierr);
  ierr = MatDestroy(&A); CHKERRQ(ierr);
  ierr = KSPDestroy(&ksp); CHKERRQ(ierr);

  // Finalize and finish
  ierr = PetscFinalize();
  return 0;
}
```

# Matlab program viewer.m for visualization of results

```
load nodes.m
load values.m
u = @(x, y) sin(2*pi*x).*sin(2*pi*y) + 1i*x.*(1 - x).*y.*(1 - y);
x_0=0.5;
y_0=0.5;
c_x= 0.1;
c_y=0.1;
epsilon = @(x, y) 2*exp(-((x-x_0).*(x-x_0)/(2*c_x.*c_x) ...
+(y-y_0).*(y-y_0)/(2*c_y.*c_y)));
% for test 2
%epsilon = @(x, y) 1+2*exp(-((x-0.5).*(x-0.5) +(y-0.7).*(y-0.7))/0.001) ...
 +  3*exp(-((x-0.2).*(x-0.2) +(y-0.6).*(y-0.6))/0.001);
n = sqrt(size(nodes, 1));
X = reshape(nodes(:, 1), n, n);
Y = reshape(nodes(:, 2), n, n);
Ur = reshape(values(:, 1), n, n);
Ui = reshape(values(:, 2), n, n);
[Xe, Ye] = meshgrid(linspace(0, 1, 30), linspace(0, 1, 30));
ur = real(u(Xe, Ye));
ui = imag(u(Xe, Ye));
Eps = epsilon(Xe, Ye)
```

```
subplot(3, 2, 1)
surf(X', Y', Ur)
title('u_h Real, computed')
  view(2)
subplot(3, 2, 2)
surf(Xe, Ye, ur)
title('u Real, exact')
  view(2)
subplot(3, 2, 3)
surf(X', Y', Ui)
title('u_h Imag, computed')
  view(2)
subplot(3, 2, 4)
surf(Xe, Ye, ui)
title('u Imag, exact')
  view(2)
subplot(3, 2, 5)
surf(Xe, Ye, Eps)
  title('exact  epsilon, 3D view')
subplot(3, 2, 6)
surf(Xe, Ye, Eps)
view(2)
title('exact  epsilon, 2D')
shg
```

# PETSc: example of Makefile for running at Chalmers

Below is presented Makefile for compilation of the program cplxmaxwell.cpp using C++/PETSc. To compile:

```
>make runmaxwell
```

To run:

```
>runmaxwell argv[1]
```

Here, argv[1] is number of the points in x and y directions (to construct mesh).

```
PETSC_ARCH=/chalmers/sw/sup64/petsc-3.10.4c
include ${PETSC_ARCH}/lib/petsc/conf/variables
include ${PETSC_ARCH}/lib/petsc/conf/rules
MPI_INCLUDE = ${PETSC_ARCH}/include/mpiuni
CXX = g++
CXXFLAGS = -Wall -Wextra -g -O0 -c -Iinclude
-I${PETSC_ARCH}/include  -I${MPI_INCLUDE}
LD = g++
LFLAGS =
OBJECTS = cplxmaxwell.o
RUNMAXWELL = runmaxwell
all: $(RUNMAXWELL)
%.o: %.cpp
$(CXX) $(CXXFLAGS) -o $@ $<
$(RUNMAXWELL): $(OBJECTS)
$(LD) $(LFLAGS) $(OBJECTS) $(PETSC_LIB) -o $@
```