Applied Numerical Linear Algebra. Lecture 5

Estimating Condition Numbers

To compute a practical error bound based on a bound (see Lecture 3)

$$\|\delta x\| = \|A^{-1}r\| \leq \|A^{-1}\|\|r\|$$

we need to estimate $||A^{-1}||$. This is also enough to estimate the condition number $k(A) = ||A^{-1}|| \cdot ||A||$, since $||A||$ is easy to compute.
One approach is to compute A^{-1} explicitly and compute its norm. However, this would cost $2n^3$, more than the original $\frac{2}{3}n^3$ for Gaussian elimination. It is a fact that most users will not bother to compute error bounds if they are expensive.
So instead of computing A^{-1} we will devise a much cheaper algorithm to *estimate* $||A^{-1}||$.

Estimating Condition Numbers

Such an algorithm is called a *condition estimator* and should have the following properties:

1. Given the L and U factors of A, it should cost $O(n^2)$, which for large enough n is negligible compared to the $\frac{2}{3}n^3$ cost of GEPP.
2. It should provide an estimate which is almost always within a factor of 10 of $||A^{-1}||$. This is all one needs for an error bound which tells you about how many decimal digits of accuracy that you have.

Estimating Condition Numbers

- There are a variety of such estimators available. We choose one to solve $Ax = b$.
- This estimator is guaranteed to produce only a lower bound on $||A^{-1}||$, not an upper bound.
- It is almost always within a factor of 10, and usually 2 to 3, of $||A^{-1}||$.
- The algorithm estimates the one-norm $||B||_1$ of a matrix B, provided that we can compute Bx and $B^T y$ for arbitrary x and y. We will apply the algorithm to $B = A^{-1}$, so we need to compute $A^{-1}x$ and $A^{-T}y$, i.e., solve linear systems. This costs just $O(n^2)$ given the LU factorization of A.

The algorithm was developed in:
W. W. Hager. Condition estimators. SIAM J. Sci. Statist. Comput., 5:311-316, 1984.
N. J. Higham. A survey of condition number estimation for triangular matrices. SIAM Rev., 29:575-596, 1987.
N. J. Higham. Experience with a matrix norm estimator. SIAM J. Sci. Statist. Comput., 11:804-809, 1990.
with the latest version in [N. J. Higham. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. ACM Trans. Math. Software].

Recall that $||B||_1$ is defined by

$$||B||_1 = \max_{x \neq 0} \frac{||Bx||_1}{||x||_1} = \max_j \sum_{i=1}^{n} |b_{ij}|.$$

It is easy to show that the maximum over $x \neq 0$ is attained at $x = e_{j_0}[0, \ldots, 0, 1, 0, \ldots, 0]^T$. (The single nonzero entry is component j_0, where $\max_j \sum_i |b_{ij}|$ occurs at $j = j_0$.)
Searching over all e_j, $j = 1, \ldots, n$, means computing all columns of $B = A^{-1}$; this is too expensive. Instead, since $||Bx||_1 = \max_{||x||_1 \leq 1} ||Bx||_1$, we can use *hill climbing* or *gradient ascent* on $f(x) \equiv ||Bx||_1$ inside the set $||x||_1 \leq 1$. $||x||_1 \leq 1$ is clearly a convex set of vectors, and $f(x)$ is a convex function, since $0 \leq \alpha \leq 1$ implies $f(\alpha x + (1-\alpha)y) = ||\alpha Bx + (1-\alpha)By||_1 \leq \alpha ||Bx||_1 + (1-\alpha)||By||_1 = \alpha f(x) + (1-\alpha) f(y)$.

Doing gradient ascent to maximize $f(x)$ means moving x in the direction of the gradient $\nabla f(x)$ (if it exists) as long as $f(x)$ increases. The convexity of $f(x)$ means $f(y) \geq f(x) + \nabla f(x) \cdot (y - x)$ (if $\nabla f(x)$ exists). To compute $\nabla f(x)$ we assume all $\sum_j b_{ij} x_j \neq 0$ in $f(x) = \sum_i \sum_j |b_{ij} x_j|$ (this is almost always true). Let $\zeta_i = sign(\sum_j b_{ij} x_j)$, so $\zeta_i = \pm 1$ and $f(x) = \sum_i \sum_j \zeta_i b_{ij} x_j$. Then $\dfrac{\partial f}{\partial x_k} = \sum_i \zeta_i b_{ik}$ and $\nabla f = \zeta^T B = (B^T \zeta)^T$.

In summary, to compute $\nabla f(x)$ takes three steps: $\omega = Bx$, $\zeta = sign(\omega)$ and $\nabla f(x) = \zeta^T B$.

ALGORITHM *Hager's condition estimator returns a lower bound $||\omega||_1$ on $||B||_1$:*

```
choose any $x$ such that $||x||_1 = 1$        /* e.g. $x_i = \frac{1}{n}$ */
repeat
$\omega = Bx$, $\zeta = sign(\omega)$, $z = B^T\zeta$,        /* $z^T = \nabla f$ */
if $||z||_\infty \leq z^T x$ then
    return $||\omega||_1$
else
    $x = e_j$ with 1 at the place $j$ where $|z_j| = ||z||_\infty$
end if
end repeat
```

Implementation in Matlab

```
x=(1/length(B))*ones(length(B),1);
iter=1;
while iter < 1000
w=B*x; xi=sign(w); z = B'*xi;
if max(abs(z)) <= z'*x
break
else
x= (max(abs(z))== abs(z));
end
iter = iter + 1;
end
LowerBound = norm(w,1);

end
```

THEOREM 1. *When $||\omega||_1$ is returned, $||\omega||_1 = ||Bx||_1$ is a local maximum of $||Bx||_1$.*
2. *Otherwise, $||Be_j||$ (at end of loop) $> ||Bx||$ (at start), so the algorithm has made progress in maximizing $f(x)$.*
Proof.
1. In this case, $||z||_\infty \le z^T x$ (*). Near x, $f(x) = ||Bx||_1 = \sum_i \sum_j \zeta_i b_{ij} x_j$ is linear in x so $f(y) = f(x) + \nabla f(x) \cdot (y - x) = f(x) + z^T(y - x)$, where $z^T = \nabla f(x)$. To show x is a local maximum we want $z^T(y - x) \le 0$ when $||y||_1 = 1$. We compute

$$\begin{array}{rcl} z^T(y-x) & = & z^T y - z^T x = \sum_i z_i \cdot y_i - z^T x \le \sum_i |z_i| \cdot |y_i| - z^T x \\ & \le & ||z||_\infty \cdot ||y||_1 - z^T x = \underbrace{||z||_\infty - z^T x}_{see(*)} \le 0. \end{array}$$

2. In this case $||z||_\infty > z^T x$. Choose $\widetilde{x} = e_j \cdot sign(z_j)$, where j is chosen so that $|z_j| = ||z||_\infty$. Then

$$\begin{array}{rcl} f(\widetilde{x}) & = & f(x) + \nabla f \cdot (\widetilde{x} - x) = f(x) + z^T(\widetilde{x} - x) \\ & = & f(x) + z^T \widetilde{x} - z^T x = f(x) + |z_j| - z^T x > f(x), \end{array}$$

where the last inequality is true by construction. □

Remarks

Higham [FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation; Experience with a matrix norm estimator] tested a slightly improved version of this algorithm by trying many random matrices of sizes 10,25,50 and condition numbers $k = 10, 10^3, 10^6, 10^9$; in the worst case the computed k underestimated the true k by a factor .44. A different condition estimator is available in Matlab as rcond. The Matlab routine cond computes the exact condition number $||A^{-1}||_2||A||_2$, it is much more expensive than rcond.

Estimating the Relative Condition Number

We can apply the Hager's algorithm to estimate the relative condition number $k_{CR}(A) = || \, |A^{-1}| \cdot |A| \, ||_\infty$ or to evaluate the bound $|| \, |A^{-1}| \cdot |r| \, ||_\infty$. We can reduce both to the same problem, that of estimating $|| \, |A^{-1}| \cdot g \, ||_\infty$, where g is a vector of nonnegative entries. To see why, let e be the vector of all ones. From definition of norm, we see that $||X||_\infty = ||Xe||_\infty$ if the matrix X has nonnegative entries. Then

$$|| \, |A^{-1}| \cdot |A| \, ||_\infty = || \, |A^{-1}| \cdot |A|e \, ||_\infty = || \, |A^{-1}| \cdot g \, ||_\infty,$$

where $g = |A|e$.

Here is how we estimate $|| \, |A^{-1}| \cdot g \, ||_\infty$. Let $G = diag(g_1, \ldots, g_n)$; then $g = Ge$. Thus

$$\begin{aligned} || \, |A^{-1}| \cdot g \, ||_\infty &= || \, |A^{-1}| \cdot Ge \, ||_\infty = || \, |A^{-1}| \cdot G \, ||_\infty = \\ &= || \, |A^{-1}G| \, ||_\infty = ||A^{-1}G||_\infty. \end{aligned} \tag{2.12}$$

The last equality is true because $||Y||_\infty = || \, |Y| \, ||_\infty$ for any matrix Y. Thus, it suffices to estimate the infinity norm of the matrix $A^{-1}G$. We can do this by applying Hager's algorithm to the matrix $(A^{-1}G)^T = GA^{-T}$, to estimate $||(A^{-1}G)^T||_1 = ||A^{-1}G||_\infty$ (see definition of norm).

Practical Error Bounds

We present two practical error bounds for our approximate solution $\widetilde{x}$ of $Ax = b$. For the first bound we use inequality $||\widetilde{x} - x||_\infty \leq ||A^{-1}||_\infty \cdot ||r||_\infty$ to get

$$error = \frac{||\widetilde{x} - x||_\infty}{||\widetilde{x}||_\infty} \leq ||A^{-1}||_\infty \cdot \frac{||r||_\infty}{||\widetilde{x}||_\infty}, \tag{2.13}$$

where $r = A\widetilde{x} - b$ is the residual. We estimate $||A^{-1}||_\infty$ by applying Algorithm to $B = A^{-T}$, estimating $||B||_1 = ||A^{-T}||_1 = ||A^{-1}||_\infty$ (see definition of norm).
Our second error bound comes from the inequality:

$$error = \frac{||\widetilde{x} - x||_\infty}{||\widetilde{x}||_\infty} \leq \frac{|||A^{-1}| \cdot |r|||_\infty}{||\widetilde{x}||_\infty}. \tag{2.14}$$

We estimate $|||A^{-1}| \cdot |r|||_\infty$ using the algorithm based on equation (2.12).

What Can Go Wrong

- Error bounds (2.13) and (2.14) are not guaranteed to provide bounds in all cases in practice.
- First, the estimate of $||A^{-1}||$ from Algorithm (or similar algorithms) provides only a lower bound, although the probability is very low that it is more than 10 times too small.
- Second, there is a small but non-negligible probability that roundoff in the evaluation of $r = A\hat{x} - b$ might make $||r||$ artificially small, in fact zero, and so also make our computed error bound too small. To take this possibility into account, one can add a small quantity to $|r|$ to account for it: the roundoff in evaluating r is bounded by

$$|(A\hat{x} - b) - fl(A\hat{x} - b)| \leq (n+1)\varepsilon(|A| \cdot |\hat{x}| + |b|), \qquad (2.15)$$

  so we can replace $|r|$ with $|r| + (n+1)\varepsilon(|A| \cdot |\hat{x}| + |b|)$ in bound (2.14) or $||r||$ with $||r|| + (n+1)\varepsilon(||A|| \cdot ||\hat{x}|| + ||b||)$ in bound (2.13).
- Third, roundoff in performing Gaussian elimination on very ill-conditioned matrices can yield such inaccurate L and U that bound (2.14) is much too low.

Improving the Accuracy of a Solution

We have just seen that the error in solving $Ax = b$ may be as large as $k(A)\varepsilon$. If this error is too large, what can we do? One possibility is to rerun the entire computation in higher precision, but this may be quite expensive in time and space. Fortunately, as long as $k(A)$ is not too large, there are much cheaper methods available for getting a more accurate solution.

Improving the Accuracy of a Solution

To solve any equation $f(x) = 0$, we can try to use Newton's method to improve an approximate solution x_i to get $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. Applying this to $f(x) = Ax - b$ yields one step of iterative refinement:

$$\begin{aligned} &r = Ax_i - b \\ &\text{solve } Ad = r \text{ for } d \\ &x_{i+1} = x_i - d \end{aligned}$$

If we could compute $r = Ax_i - b$ exactly and solve $Ad = r$ exactly, we would be done in one step, which is what we expect from Newton applied to a linear problem. Roundoff error prevents this immediate convergence. The algorithm is interesting and of use precisely when A is so ill-conditioned that solving $Ad = r$ (and $Ax_0 = b$) is rather inaccurate.

Suppose that r is computed in double precision and $k(A) \cdot \varepsilon < c \equiv \frac{1}{3n^3 g+1} < 1$ where n is the dimension of A and g is the pivot growth factor. Then repeated iterative refinement converges with

$$\frac{||x_i - A^{-1}b||_\infty}{||A^{-1}b||_\infty} = O(\varepsilon).$$

Note that the condition number does not appear in the final error bound. This means that we compute the answer accurately independent of the condition number, provided that $k(A)\varepsilon$ is sufficiently less than 1.(In practice, c is too conservative an upper bound, and the algorithm often succeeds even when $k(A)\varepsilon > c$.)

For partial pivoting of $n \times n$ matrices $g \leq 2^{n-1}$. The classical definition used by Wilkinson is (k is the number of permut.):

$$g(A) := \frac{\max_{i,j,k} |a_{ij}|^{(k)}}{\max_{i,j} |a_{ij}|}.$$

Another definition for LU decomposition of A is:

$$g(A) := \frac{\||L| \cdot |U|\|_\infty}{\|A\|_\infty}.$$

Single Precision Iterative Refinement

THEOREM.
Suppose that r is computed in single precision and

$$||A^{-1}||_\infty \cdot ||A||_\infty \cdot \frac{\max_i(|A| \cdot |x|)_i}{\min_i(|A| \cdot |x|)_i} \cdot \varepsilon < 1.$$

Then one step of iterative refinement yields x_1 such that $(A + \delta A)x_1 = b + \delta b$ with $|\delta a_{ij}| = O(\varepsilon)|a_{ij}|$ and $|\delta b_i| = O(\varepsilon)|b_i|$. In other words, the componentwise relative backward error is as small as possible. For example, this means that if A and b are sparse, then δA and δb have the same sparsity structures as A and b, respectively.

For a proof, see

N. J. Higham. Accuracy and Stability of Numerical Algorithms. SIAM, Philadelphia, PA, 1996.

M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. SIAM J. Matrix Anal. AppL, 10:165-190, 1989.

R. D. Skeel. Scaling for numerical stability in Gaussian elimination. Journal of the ACM, 26:494-526, 1979.

R. D. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. Math. Comp., 35:817-832, 1980.

R. D. Skeel. Effect of equilibration on residual size for partial pivoting. SIAM J. Numer. Anal, 18:449-454, 1981.

Single precision iterative refinement and the error bound (2.14) are implemented in LAPACK routines like sgesvx.

Equilibration

- There is one more common technique for improving the error in solving a linear system: **equilibration**. This refers to choosing an appropriate diagonal matrix D and solving $D\mathbf{Ax} = Db$ instead of $Ax = b$. D is chosen to try to make the condition number of DA smaller than that of A.
- For instance, choosing d_{ii} to be the reciprocal of the two-norm of row i of A would make DA nearly equal to the identity matrix, reducing its condition number from 10^{14} to 1.
- It is possible to show that choosing D this way reduces the condition number of DA to within a factor of $\sqrt{n}$ of its smallest possible value for any diagonal D [A. Van Der Sluis. Condition numbers and equilibration of matrices. Numer. Math., 14:14-23, 1969].
- In practice we may also choose two diagonal matrices D_{row} and D_{col} and solve $(D_{row}A\mathbf{D_{col}})\bar{\mathbf{x}} = D_{row}b$, $x = D_{col}\bar{x}$ and thus $D_{row}Ax = D_{row}b$.

Special Linear Systems

It is important to exploit any special structure of the matrix to increase speed of solution and decrease storage. We will consider only real matrices:

- s.p.d. matrices,
- symmetric indefinite matrices,
- band matrices,
- general sparse matrices,
- dense matrices depending on fewer than n^2 independent parameters.

Real Symmetric Positive Definite Matrices

Recall that a real matrix A is s.p.d. if and only if $A = A^T$ and $x^T Ax > 0$ for all $x \neq 0$. In this section we will show how to solve $Ax = b$ in half the time and half the space of Gaussian elimination when A is s.p.d.

PROPOSITION.

1. *If X is nonsingular, then A is s.p.d. if and only if $X^T AX$ is s.p.d.*
2. *If A is s.p.d. and H is any principal submatrix of $A(H = A(j : k, j : k)$ for some $j \leq k)$, then H is s.p.d.*
3. *A is s.p.d. if and only if $A = A^T$ and all its eigenvalues are positive.*
4. *If A is s.p.d., then all $a_{ii} > 0$, and $\max_{ij} |a_{ij}| = \max_i a_{ii} > 0$.*
5. *A is s.p.d. if and only if there is a unique lower triangular nonsingular matrix L, with positive diagonal entries, such that $A = LL^T$. $A = LL^T$ is called the Cholesky factorization of A, and L is called the Cholesky factor of A.*

Proof.
1. **If X is nonsingular, then A is s.p.d. if and only if X^TAX is s.p.d.**
X nonsingular implies $Xx \neq 0$ for all $x \neq 0$, so $x^TX^TAXx > 0$ for all $x \neq 0$. So A s.p.d. implies X^TAX is s.p.d. Use X^{-1} to deduce the other implication.

2. **If A is s.p.d. and H is any principal submatrix of $A(H = A(j:k, j:k)$ for some $j \leq k)$, then H is s.p.d.** Suppose first that $H = A(1:m, 1:m)$. Then given any m-vector y, the n-vector $x = [y^T, O]^T$ satisfies $y^T Hy = x^T Ax$. So if $x^T Ax > 0$ for all nonzero x, then $y^T Hy > 0$ for all nonzero y, and so H is s.p.d. If H does not lie in the upper left corner of A, let P be a permutation so that H does lie in the upper left corner of $P^T AP$ and apply Part 1.

3. A **is s.p.d. if and only if** $A = A^T$ **and all its eigenvalues are positive.**
Let X be the real, orthogonal eigenvector matrix of A so that $X^T AX = \bigwedge$ is the diagonal matrix of real eigenvalues λ_i. Since $x^T \bigwedge x = \sum_i \lambda_i x_i^2$, $\bigwedge$ is s.p.d. if and only if each $\lambda_i > 0$. Now apply Part 1.

4. **If** A **is s.p.d., then all** $a_{ii} > 0$**, and** $\max_{ij} |a_{ij}| = \max_i a_{ii} > 0$**.** Let e_i be the ith column of the identity matrix. Then $e_i^T A e_i = a_{ii} > 0$ for all i. If $|a_{kl}| = \max_{ij} |a_{ij}|$ but $k \neq l$, choose $x = e_k - sign(a_{kl}) e_l$. Then $x^T A x = a_{kk} + a_{ll} - 2|a_{kl}| \leq 0$, contradicting positive-definiteness.

5. A **is s.p.d. if and only if there is a unique lower triangular nonsingular matrix** L**, with positive diagonal entries, such that** $A = LL^T$**.** $A = LL^T$ **is called the Cholesky factorization of** A**, and** L **is called the Cholesky factor of** A**.**
Suppose $A = LL^T$ with L nonsingular. Then $x^T Ax = (x^T L)(L^T x) = ||L^T x||_2^2 > 0$ for all $x \neq 0$, so A is s.p.d. If A is s.p.d., we show that L exists by induction on the dimension n. If we choose each $l_{ii} > 0$, our construction will determine L uniquely. If $n = 1$, choose $l_{11} = \sqrt{a_{11}}$, which exists since $a_{11} > 0$. As with Gaussian elimination, it suffices to understand the block 2-by-2 case.

Write

$$\begin{aligned} A &= \begin{bmatrix} a_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} \\ &= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{A_{12}^T}{\sqrt{a_{11}}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \frac{A_{12}}{\sqrt{a_{11}}} \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & A_{12} \\ A_{12}^T & \tilde{A}_{22} + \frac{A_{12}^T A_{12}}{a_{11}} \end{bmatrix}, \end{aligned}$$

so the $(n-1)$-by-$(n-1)$ matrix $\tilde{A}_{22} + \frac{A_{12}^T A_{12}}{a_{11}}$ is symmetric.

We note to the previous slide (here we write how we can obtain elements in the last matrix on the previous slide) that

$$\begin{aligned} A &= \begin{bmatrix} a_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix} = LL^T = \\ &= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ y & \tilde{L}_{22} \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & y^T \\ 0 & \tilde{L}_{22}^T \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & \sqrt{a_{11}}y^T \\ \sqrt{a_{11}}y & yy^T + \tilde{L}_{22}\tilde{L}_{22}^T \end{bmatrix}, \end{aligned}$$

and thus $A_{12} = \sqrt{a_{11}}y$ such that we can find $y = \frac{A_{12}}{\sqrt{a_{11}}}$ and $A_{22} = yy^T + \tilde{L}_{22}\tilde{L}_{22}^T = \tilde{A}_{22} + \frac{A_{12}^T A_{12}}{a_{11}}$ with $\tilde{A}_{22} = \tilde{L}_{22}\tilde{L}_{22}^T$.

By Part 1 above, $\begin{bmatrix} 1 & 0 \\ 0 & \tilde{A}_{22} \end{bmatrix}$ is s.p.d, so by Part 2 $\tilde{A}_{22}$ is s.p.d.

Thus by induction there exists an $\tilde{L}$ such that $\tilde{A}_{22} = \tilde{L}\tilde{L}^T$ and

$$\begin{aligned} A &= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{A_{12}^T}{\sqrt{a_{11}}} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \tilde{L}\tilde{L}^T \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \frac{A_{12}}{\sqrt{a_{11}}} \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} \sqrt{a_{11}} & 0 \\ \frac{A_{12}^T}{\sqrt{a_{11}}} & \tilde{L} \end{bmatrix} \begin{bmatrix} \sqrt{a_{11}} & \frac{A_{12}}{\sqrt{a_{11}}} \\ 0 & \tilde{L}^T \end{bmatrix} \equiv LL^T. \ \square \end{aligned}$$

We may rewrite this induction as the following algorithm.

ALGORITHM *Cholesky algorithm:*

```
for $j = 1$ to $n$
    $l_{jj} = (a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2)^{1/2}$
    for $i = j + 1$ to $n$
        $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}) / l_{jj}$
    end for
end for
```

If A is not positive definite, then (in exact arithmetic) this algorithm will fail by attempting to compute the square root of a negative number or by dividing by zero; this is the cheapest way to test if a symmetric matrix is positive definite.

The number of flops in Cholesky algorithm

In Cholesky algorithm L can overwrite the lower half of A. Only the lower half of A is referred to by the algorithm, so in fact only $n(n+l)/2$ storage is needed instead of n^2. The number of flops is

$$\textbf{Number of operations in Cholesky algorithm} = \sum_{j=1}^{n}\left(2j + \sum_{i=j+1}^{n} 2j\right) = \frac{1}{3}n^3 + O(n^2). \tag{1}$$

The number of operations for LU decomposition is $\frac{2}{3}n^3 + O(n^2)$. Pivoting is not necessary for Cholesky to be numerically stable. We show this as follows. The same analysis as for Gaussian elimination reveals that we will have similar formula for error E in Cholesky decomposition as in the LU decomposition:

$$A = LL^T + E,$$

where error in Cholesky decomposition will be bounded as

$$|E| \leq n\epsilon |L| \cdot |L^T|.$$

Example: solution of Poisson's equation

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

$$\begin{aligned} -\triangle u(x) &= f(x) \ \text{ in } \ \Omega, \\ u &= 0 \text{ on } \partial\Omega. \end{aligned} \tag{2}$$

Here $f(x)$ is a given function, $u(x)$ is the unknown function, and the domain Ω is the unit square $\Omega = \{(x_1, x_2) \in (0,1) \times (0,1)\}$. To solve numerically (2) we first discretize the domain Ω with $x_{1_i} = ih_1$ and $x_{2_j} = jh_2$, 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 $0 < i, j < n + 1$ 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:

$$n_{glob} = j + n_i(i - 1). \tag{3}$$

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

$$\Delta u_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}, \qquad (4)$$

where $u_{i,j}$ is the solution at the discrete point (i,j). Using (4), we obtain the following scheme for solving problem (2):

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

where $f_{i,j}$ are the value of the function f at the discrete point (i,j). Then (5) can be rewritten as

$$-\left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j} + u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) = h^2 f_{i,j}, \qquad (6)$$

or in the more convenient form as

$$-u_{i+1,j} + 4u_{i,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = h^2 f_{i,j}. \qquad (7)$$

System (7) 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

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

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 Ω as described above with $N = n_i = n_j = 3$. We present the schematic discretization via the global nodes numbering for all $1 \leq i, j < n + 1$

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

in the following scheme:

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

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

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

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

We illustrate the numerical solution of problem (2). We define the right hand side $f(x)$ of (2) as

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

The coefficient $a(x_1, x_2)$ in (9) is given by the following Gaussian function:

$$a(x_1, x_2) = 1 + A \exp\left(-\frac{(x_1 - c_1)^2}{2s_1^2} - \frac{(x_2 - c_2)^2}{2s_2^2}\right), \qquad (10)$$

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_{1i}, x_{2j}) such that $x_{1i} = ih, x_{2j} = jh$ 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

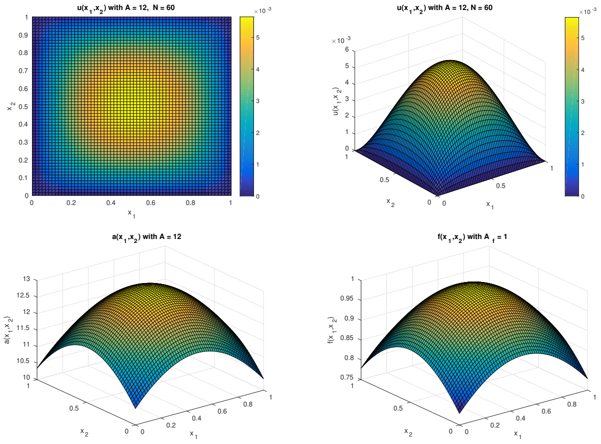


Figure: Solution of Poisson's equation (2) with $f(x_1, x_2)$ as in (9) and $a(x_1, x_2)$ as in (10).

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

$$f(x_1,x_2) = 1 + 10e^{\left(-\frac{(x_1-0.25)^2}{0.02} - \frac{(x_2-0.25)^2}{0.02}\right)} + 10e^{\left(-\frac{(x_1-0.75)^2}{0.02} - \frac{(x_2-0.75)^2}{0.02}\right)} \quad (11)$$

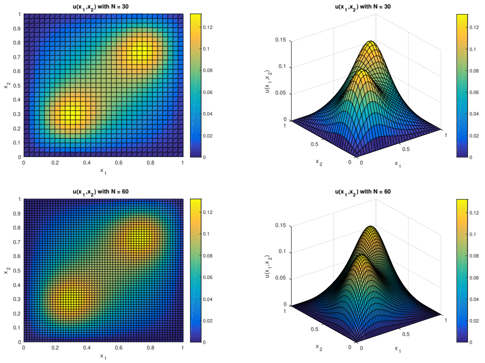


Figure: Solution of Poisson's equation (2) with $f(x_1,x_2)$ as in (11).

Computer exercise 4 (3 p.)

This exercise can be viewed as a training example for Master's project "Efficient implementation of Helmholtz equation with applications in medical imaging", see Master's projects homepage and Lecture 1. Solve the Helmholtz equation

$$\begin{aligned} \Delta u(x,\omega) + \omega^2 \varepsilon'(x) u(x,\omega) &= i\omega J, \\ \lim_{|x|\to\infty} u(x,\omega) &= 0. \end{aligned} \tag{12}$$

in two dimensions using PETSC. Here, $\varepsilon'(x)$ is the spatially distributed complex dielectric function which can be expressed as

$$\varepsilon'(x) = \varepsilon_r(x)\frac{1}{c^2} - i\mu_0\frac{\sigma(x)}{\omega}, \tag{13}$$

where $\varepsilon_r(x) = \varepsilon(x)/\varepsilon_0$ and $\sigma(x)$ are the dimensionless relative dielectric permittivity and electric conductivity functions, respectively, ε_0, μ_0 are the permittivity and permeability of the free space, respectively, and $c = 1/\sqrt{\varepsilon_0\mu_0}$ is the speed of light in free space, and ω is the angular frequency.

Take appropriate values for ω, ε', J. For example, take

$$\omega = \{40, 60, 80, 100\}, \varepsilon_r = \{2, 4, 6\}; \sigma = \{5, 0.5, 0.05\}, J = 1.$$

Hints (more information see on the course page):

1. Study Example 12.5 of [BKK] where is presented solution of the Dirichlet problem for the Poisson's equation using PETSc. PETSc programs for solution of this problem are available download from the course homepage: go to the link of the book [BKK] and click to "GitHub Page with MATLAB® Source Codes" on the bottom of this page.
2. Modify PETSc code of the Example 12.5 of [BKK], or PETSc code on the course homepage (see it at the end of this lecture) such that the equation (12) can be solved. Note that solution of the equation (12) is complex. You should include in PETSc code

   ```
   #include <complex>
   ```

   to be able work with complex numbers in C++.

Below is an example of definition of the complex array in C++ and assigning values to the real and imaginary parts:

```
complex<double> *complex2d = new complex<double>[nno];
double a = 5.4;
double b = 3.1;
for (int i=0; i < nno; i++)
complex2d[i].real() = a;
complex2d[i].imag() = b;

delete[] complex2d;
```

Example of the definition of the complex right hand side in PETSc is presented below:

```
PetscScalar rhs(const PetscReal x, const PetscReal y)

PetscReal realpart, imagpart;
PetscReal pi = 3.14159265359;
realpart = pi*sin(2*pi*x)*cos(2*pi*y);
imagpart = x*x + y*y;
PetscScalar f(rpart, ipart);
return f;
```

3. Example of Makefile for running C++/PETSc code with real numbers at Chalmers is presented in Example 12.5 of [BKK].

   Further information about PETSc can be found on the link:

   https://www.mcs.anl.gov/petsc/

4. Choose the two-dimensional convex computational domain Ω as you prefer. For example, $\Omega = [0,1] \times [0,1]$. Choose boundary condition at the boundary of $\partial\Omega$ such that the condition $\lim_{|x| \to \infty} u(x, \omega) = 0$ is satisfied, for example, take some functions in the form of Gaussian exp^{-x^2}.

5. Take appropriate values for ω, ε', J. Analyze obtained results for different $\omega, \varepsilon_r, \sigma, J$. For example, take

$$\omega = \{40, 60, 80, 100\}, \varepsilon_r = \{2, 4, 6\}; \sigma = \{5, 0.5, 0.05\}, J = 1.$$

6. Values of c, μ_0, ε_0 in (13) are known constants.

- Vacuum permittivity, sometimes called the electric constant ε_0 and measured in F/m (farad per meter):

$$\varepsilon_0 \approx 8.85 \cdot 10^{-12}$$

- The permeability of free space,or the magnetic constant μ_0 measured in H/m (henries per meter):

$$\mu_0 \approx 12.57 \cdot 10^{-7}$$

- The speed of light in a free space is given by formula $c = 1/\sqrt{\varepsilon_0 \mu_0}$ and is measured in m/c (metres per second):

$$c \approx 300\ 000\ 000$$

Example of the Makefile to compile PETSc program with complex numbers is presented below:

```
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 = Program.o
RUN = runprogram
all: $(RUN)
$(CXX) $(CXXFLAGS) -o $@ $<
$(RUN): $(OBJECTS)
$(LD) $(LFLAGS) $(OBJECTS) $(PETSC_LIB) -o $@
```

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:

$$\begin{aligned} \triangle u(x) + \omega^2 \varepsilon(x) u = f(x) \ \text{ in } \ \Omega, \\ u = 0 \text{ on } \partial\Omega. \end{aligned} \tag{14}$$

Here $f(x)$ is a given function, $u(x)$ is the unknown function to be computed, and the domain Ω is the unit square $\Omega = \{(x_1, x_2) \in (0,1) \times (0,1)\}$.
The exact solution of (14) with the right hand side

$$\begin{aligned} f(x_1,x_2) = -(8\pi^2)\sin(2\pi x_1)\sin(2\pi x_2) - 2ix_1(1-x_1) - 2ix_2(1-x_2) \\ + \omega^2\varepsilon(x)(\sin(2\pi x_1)\sin(2\pi x_2) + ix_1(1-x_1)x_2(1-x_2)) \end{aligned} \tag{15}$$

is the function

$$u(x_1,x_2) = \sin(2\pi x_1)\sin(2\pi x_2) + ix_1(1-x_1)x_2(1-x_2). \tag{16}$$

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

$$P^{-1}Ax = P^{-1}b. \tag{17}$$

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 (17) by $P^{1/2}$ to get

$$(P^{-1/2}AP^{-1/2})(P^{1/2}x) = P^{-1/2}b. \tag{18}$$

Preconditioning for Linear Systems

We note that the system (18) 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 (18) 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: $r = 0$;  $x_0 = 0$;  $R_0 = b$;  $p_1 = P^{-1}b; y_0 = P^{-1}R_0$
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

$$P = \begin{pmatrix} P_{1,1} & ... & 0 \\ ... & ... & ... \\ 0 & ... & P_{r,r} \end{pmatrix} \tag{19}$$

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

$$A = \begin{pmatrix} A_{1,1} & ... & A_{1,r} \\ ... & ... & ... \\ A_{r,1} & ... & A_{r,r} \end{pmatrix} \tag{20}$$

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 $A = D + L + L^T$ then the SSOR preconditioner matrix is defined as
$$P = (D + L)D^{-1}(D + L)^T$$
It can also be parametrised by ω 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 (14) 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

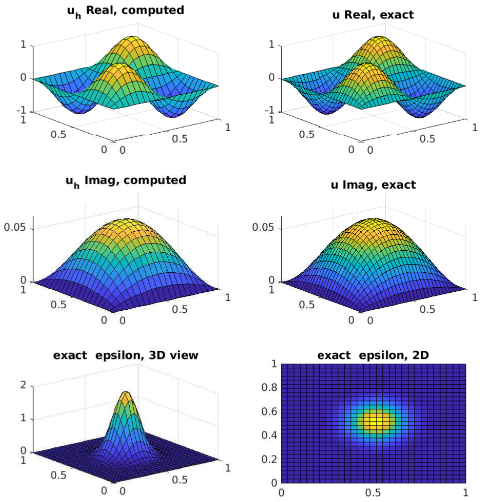
```
nodes.m
values.m
```

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 (14) 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
```