

# MMATH PROJECT

**Deadline Noon January 11th 2007**

Consider the boundary value problem: Find  $u(x, y)$  such that

$$-\Delta u = 1 \quad \text{in} \quad \Omega := (0, 1) \times (0, 1)$$

$$u = 0 \quad \text{on} \quad \partial\Omega := \text{Boundary of } \Omega$$

The solution  $u$  is to be approximated by a sequence of functions  $u^n$ , where  $n$  relates to the size of a grid placed on the domain  $\Omega$ . That is, an  $n$  by  $n$  uniform grid is placed on  $\Omega$ . At every grid point  $(x_i, y_j)$ ,  $i, j = 1, 2, \dots, n+1$  we wish to find an approximation  $u^n(x_i, y_j)$  to  $u(x_i, y_j)$ . Using a finite difference approximation we obtain a sequence of linear systems to solve of the form: Find  $\mathbf{u}_n \in \mathbb{R}^{n-1}$  such that

$$A_n \mathbf{u}_n = \mathbf{b}_n, \quad A_n \in \mathbb{R}^{(n-1) \times (n-1)}, \quad \mathbf{b}_n \in \mathbb{R}^{n-1},$$

where  $\mathbf{u}_n$  is the values  $u^n(x_i, y_j)$ ,  $i, j = 2, 3, \dots, n-1$ , in some order,  $\mathbf{b}_n$  has entries 1 corresponding to the right hand side of the initial problem.

We will firstly look at how using a dense matrix compares to using a sparse one.

1. For a given  $n$  the matrix  $A_n$  has dimension  $(n-1) \times (n-1)$ . The following function will create the dense matrix  $A_n$ :

```
function [A] = build_A_matrix(n)

    A = zeros(n*n, n*n);

    %% diagonal
    A = A+diag(4*ones(n^2,1));

    %% entry above diagonal
    d = -ones(n^2-1,1);
    d(n:n:n^2-n) = 0;
    A = A+diag(d,1);

    %% entry below diagonal
    A = A+diag(d, -1);

    %% second super and sub diagonals
    d = ones(n^2-n,1);
    A = A - diag(d,n);
    A = A - diag(d,-n);

    %% Scale the matrix
    A = n^2*A;

return
```

The following function plots the discrete approximation for a given  $n$ .

```
function [U,X,Y] = twod_plot(N,u)

x = [1/N:1/N:1]; y = x;
[X,Y] = meshgrid(x,y);

ncount = 1;

for i = 1:N
    for j = 1:N

        U(i,j) = u(ncount);
        ncount = ncount+1;
    end
end
figure(1)
surf(X,Y,U); shading flat;
```

2. Investigate how Gaussian Elimination performs (cputime) for solving the above problem as  $n$  increases.
3. Investigate how Gauss-Seidel performs (cputime) for solving the above problem as  $n$  increases with a tolerance of  $10^{-6}$ .
4. Investigate how Conjugate Gradients performs (cputime) for solving the above problem as  $n$  increases with a tolerance of  $10^{-6}$ .
5. Repeat the above three questions but firstly convert each  $A_n$  into a sparse matrix. This can be done using the following software:

```
function [A] = build_sparse_A_matrix(n)

    %% Get index entries for a given row
    I = [-n,-1,0,1,n];

    %% get value for given entries
    e = ones(n^2,1);
    d_1 = e;
    d_2 = e;
    d_1(n:n:n^2) = 0;
    d_2(n+1:n:n^2) = 0;
    d = n^2*[-e,-d_1,4*e,-d_2,-e];

    %% make sparse matrix
    A = spdiags(d,I,n^2,n^2);

return
```

We can now repeat the above using much larger matrices.

6. Draw conclusions.

We now want to investigate the idea of preconditioning the matrix  $A_n$  so that the conjugate gradients improve. The basic idea is to use a matrix  $M_n = P_n P_n^T$  such that  $M_n$  is easily inverted and

$$\kappa(P_n^{-1} A_n P_n^{-T}) < \kappa(A_n),$$

where  $\kappa(\cdot)$  denotes the condition number of a matrix.

So we are left to solve the equivalent problem: Find  $\mathbf{u}_n$  such that

$$P_n^{-1}A_nP_n^{-1}(P_n\mathbf{u}_n) = P_n^{-1}\mathbf{b}_n.$$

1. Describe why the above preconditioner is desirable.
2. The following describes the steps required for preconditioned conjugate gradients:

$$\mathbf{x}_0 = \mathbf{0}, \quad \mathbf{r}_0 = \mathbf{b}, \quad \mathbf{p}_0 = M^{-1}\mathbf{r}_0, \quad \mathbf{z}_0 = \mathbf{p}_0$$

For  $n = 1, 2, 3, \dots$

$$\alpha_n = (\mathbf{r}_{n-1}^T \mathbf{z}_{n-1}) / (\mathbf{p}_{n-1}^T A \mathbf{p}_{n-1}),$$

$$\mathbf{x}_n = \mathbf{x}_{n-1} + \alpha_n \mathbf{p}_{n-1},$$

$$\mathbf{r}_n = \mathbf{r}_{n-1} - \alpha_n A \mathbf{p}_{n-1},$$

$$\mathbf{z}_n = M^{-1} \mathbf{r}_n,$$

$$\beta_n = (\mathbf{r}_n^T \mathbf{z}_n) / (\mathbf{r}_{n-1}^T \mathbf{z}_{n-1}),$$

$$\mathbf{p}_n = \mathbf{z}_n + \beta_n \mathbf{p}_{n-1}$$

Write a Matlab routine to do this.

3. We are now left to find a suitable preconditioner. We will consider the incomplete Cholesky factorisation, type `help cholinc` in Matlab. We will use this function to construct the preconditioner

$$M = R^T * R.$$

In your software you should always use

$$M^{-1} = R \setminus (R' \setminus \mathbf{r});$$

describe the incomplete Cholesky factorisation and investigate what happens as we vary the *drop tolerance*, say  $\tau$ , in the factorisation. i.e. construct an incomplete Cholesky factorisation, using `cholinc` with a drop tolerance of  $\tau = 1$

```
>> R = cholinc(A,1);
```

Use it as a preconditioner, then reduce  $\tau$ , say by an order of magnitude and run again. Look at number of iterations and `cputime`. You should do this with large sparse matrices and look also at the trend for a given drop tolerance with respect to mesh size.

4. Finally, summarise the project and draw conclusions.