THE COLLEGE OF WILLIAM AND MARY

Programming Assignment #1

Alexander Powell

March 19, 2014

1

The code below shows the MATLAB implementation of the preconditioned conjugate gradient method. Note that both precondioners are written in the code but one is commented out.

```
function [ ] = pcg( n, A, b, x )
% This function implements the preconditioned conjugate gradient method
% Note: the function itself doesn't have any explicit outputs, only
% print stmts.
TOL = 1e-15;
N = 10000;

% step 1
r = b - A * x;

% This is where to change the conditioner:
cond_inv = 4 * eye(n);          % M1

%ident = T_mat(n/2);            % M2
%zer = zeros(n/2);
%cond_inv = [ident, zer; zer, ident];

w = cond_inv * r;
v = cond_inv' * w;
alpha = 0;
```

```
for j = 1:n
    alpha = alpha + (w(j))^2;
end

% step 2
k = 1;
% step 3
while k <= N
    % step 4
    if norm(v) < TOL
        fprintf('The solution vector is: %d\n', x);
        fprintf('\n');
        fprintf('And the residual is : %d\n', r);
        fprintf('\n');
        fprintf('Number of iterations is: %d\n', k);
        fprintf('The procedure was successful. \n');
        break;
    end

    % step 5
    u = A * v;
    bottom = 0;
    for j = 1:n
        bottom = bottom + v(j) * u(j);
    end
    t = alpha / bottom;
    x = x + t * v;
    r = r - t * u;
    w = cond_inv * r;
    beta = 0;
    for j = 1:n
        beta = beta + (w(j))^2;
    end

    % step 6
    if abs(beta) < TOL
        if norm(r) < TOL
            fprintf('The solution vector is: %d\n', x);
            fprintf('\n');
            fprintf('And the residual is : %d\n', r);
            fprintf('\n');
            fprintf('Number of iterations is: %d\n', k);
            fprintf('The procedure was successful. \n');
            break;
```

```
        end
    end

    % step 7
    s = beta / alpha;
    v = cond_inv' * w + s * v;
    alpha = beta;
    k = k + 1;
end

% step 8
if k > N
    fprintf('Number of iterations is: %d\n', k);
    fprintf('The maximum number of iterations was exceeded. \n');
    fprintf('The procedure was unsuccessful. \n');
end
end
```

Also, to calculate the second preconditioner ($M2$), I wrote a separate MATLAB function displayed below.

```
function [ A ] = T_mat( n )
% This function generates the A_n^2xn^2 matrix asked for in the project.
mat = zeros(n);
for i = 1:n
    for j = 1:n
        if i == j
            mat(i,j) = 4;
        elseif i == (j + 1) || i == (j - 1)
            mat(i,j) = -1;
        else
            mat(i,j) = 0;
        end
    end
end
A = mat;
end
```

2

The following are tables displaying the number of iterations and execution times of different algorithms. Also, it should be noted that the ∞ symbol in certain cells indicates that the algorithm took too long to reasonably solve the system of the machine running the program ran out of memory. Also, the execution times were found using MATLAB's built-in tic; and toc; methods.

Algorithm Performance for n=10

	Execution Time (sec)	Num of Iterations
Gauss Elim	0.00062	N/A
Jacob	3.266225	846
G-S	53.075348	420
CG	0.00195	15
PCG - M1	0.005567	17
PCG - M2	0.006627	69

Algorithm Performance for n=20

	Execution Time (sec)	Num of Iterations
Gauss Elim	0.001311	N/A
Jacob	61.60206	3024
G-S	∞	∞
CG	∞	∞
PCG - M1	0.018958	52
PCG - M2	0.053857	228

Algorithm Performance for n=40

	Execution Time (sec)	Num of Iterations
Gauss Elim	0.003146	N/A
Jacob	1780.983	11235
G-S	∞	∞
CG	0.009074	79
PCG - M1	0.064816	117
PCG - M2	1.577816	515

Algorithm Performance for n=80

	Execution Time (sec)	Num of Iterations
Gauss Elim	0.017373	N/A
Jacob	∞	∞
G-S	∞	∞
CG	0.242193	162
PCG - M1	0.269893	258
PCG - M2	∞	∞

Algorithm Performance for n=160

	Execution Time (sec)	Num of Iterations
Gauss Elim	0.06052	N/A
Jacob	∞	∞
G-S	∞	∞
CG	0.176183	329
PCG - M1	1.503575	521
PCG - M2	∞	∞

The preconditioner I came up with was to use the inverse of $M2$. From the results, you can see that it performs much better than $M2$ but not as well as $M1$.

Algorithm Performance of PCG with new preconditioner

	Execution Time (sec)	Num of Iterations
n = 10	0.025841	41
n = 20	0.040249	70
n = 40	0.632577	133
n = 80	0.820794	257
n = 160	1.485363	523

So, judging from the results in the tables above, certain algorithms do a much better job than others. The slowest methods were the iterative techniques of Jacobian and Gauss-Seidel methods. As far as the methods that I implemented, the conjugate gradient was probaby the fastest. The preconditioned conjugate gradient method also worked well depending on which preconditioner was used. The best preconditioner was $M_1 = diag(4)$, but taking the inverse of M_1 worked well too. However, the preconditioner M_2 took significantly longer to run and required more iterations. Surprisingly, Gaussian elimination, was the fastest. However, since we were relying on MATLAB's A \b method and we can't see the code behind it, we can't be certain what's going on behind the scenes; it's possible that MATLAB has some function calls to optimize the process which would explain the quick execution times.

The following code is the implementation of just the conjugate gradient method without the preconditioner.

```
function [ x ] = cg( A, b, x )

TOL = 1e-15;
N = 10000;

r = b - A * x;
p = r;
r_old = r' * r;
```

```
k = 1;
while k <= N
    prod = A * p;
    denom = p' * prod;
    alpha = r_old / denom;
    x = x + alpha * p;
    r = r - alpha * prod;
    r_new = r' * r;
    if r_new <= TOL
        fprintf('Algorithm complete after %d iterations: \n', k);
        break;
    end
    p = r + r_new / r_old * p;
    r_old = r_new;
    k = k + 1;
end

if k > N
    fprintf('The maximum number of iterations was exceeded. \n');
end
end
```

The following code shows how to compute the Jacobian Iterative method.

```
function [  ] = jb( n, A, b )
% This functions performs the jacobian iterative method to solve a system
% of linear equations.

XO = zeros(n,1);
TOL = 1e-15;
N = 1000;

%step 1
k = 1;

%step 2
null_x = zeros(n,1);
while k <= N
    %step 3
    for i = 1:n
        empty = 0;
        for j = 1:n
            if (j ~= i)
```

```
                empty = empty + (A(i,j) * XO(j));
            end
        end
        null_x(i) = (1/A(i,i)) * (-1 * empty + b(i));
    end

    %step 4
    norm = null_x - XO;
    norm = abs(norm);
    final_norm = max(norm);
    if final_norm < TOL
        fprintf('The procedure was successful. \n');
        fprintf('The solution vector is: \n');
        fprintf('\n');
        fprintf('%d\n', null_x);
        fprintf('\n');
        fprintf('After K iterations: %d\n', k);
        break;
    end

    k = k + 1;

    for i = 1:n
        XO(i) = null_x(i);
    end
end

if k > N
    fprintf('The procedure was unsuccessful. \n');
    fprintf('Maximum number of iterations exceeded. \n');
end
end
```

The following code shows how to compute the Gauss-Seidel Iterative method.

```
function x = gs( A, b, x0 )
    tol = 1e-15;
    N = 10000;
    n = length(b);
    x = zeros(n, 1);
    for k = 1:N
        for i = 1:n
            s = 0;
```

```
            cond = true;
            for j = 1:n
                if j == i
                    cond = false;
                elseif cond
                    s = s + A(i, j) * x(j);
                else
                    s = s + A(i, j) * x0(j);
                end
            end
            x(i) = (-s + b(i)) / A(i, i);
        end
        if max(abs(x - x0)) < tol
            fprintf('required %d iterations\n', k);
            return;
        end
        x0 = x;
    end
    fprintf('required %d iterations\n', k);
end
```