

- Before you may validly use CG on a matrix, you have to check that it is s.p.d. In particular, convergence of CG is not sufficient to show spd'ness.

1. Problems 7 and 8, p. 248

Results:

Problem 7.

A =

```
  7  -3  0  0  0
 -3  9  1  0  0
  0  1  3 -1  0
  0  0 -1 10  4
  0  0  0  4  6
```

b =

```
 4
-6
 3
 7
 2
```

```
CG took          5 iterations.
SOR took         9 iterations.
Gauss-Seidel took 14 iterations.
Jacobi took      27 iterations.
```

Problem 8.

A =

```
 4 -1  0 -1  0  0
-1  4 -1  0 -1  0
 0 -1  4  0  0 -1
-1  0  0  4 -1  0
 0 -1  0 -1  4 -1
 0  0 -1  0 -1  4
```

b =

```
-1
 0
 1
-2
 1
 2
```

```
CG took          6 iterations.
SOR took        10 iterations.
Gauss-Seidel took 16 iterations.
```

Jacobi took 27 iterations.

Discussion: It's quite apparent that the Conjugate Gradient Method beats the living daylights out of any other method, even optimally-parametrized SOR. Also note that the ranking is quite predictable and does not change much between the two problems.

Code:

```
function p1
    A = [...
        7,-3,0,0,0;...
        -3,9,1,0,0;...
        0,1,3,-1,0;...
        0,0,-1,10,4;...
        0,0,0,4,6];

    b = [4;-6;3;7;2];

    disp('Problem 7.')
    disp('-----')
    run_comparison(A, b, 1.0923)

    disp('')
    disp('Problem 8.')
    disp('-----')
    A = [...
        4,-1,0,-1,0,0;...
        -1,4,-1,0,-1,0;...
        0,-1,4,0,0,-1;...
        -1,0,0,4,-1,0;...
        0,-1,0,-1,4,-1;...
        0,0,-1,0,-1,4];

    b = [-1;0;1;-2;1;2];

    run_comparison(A, b, 1.1128)
end
% -----
function run_comparison(A, b, sor_optimal_omega)
    A
    b
    tol = 5e-7;
    [x_cg, itcount_cg] = it_cg(A, b, tol);
    [x_jacobi, itcount_jacobi] = it_jacobi(A, b, tol);
    [x_gs, itcount_gs] = it_gauss_seidel(A, b, tol);
    [x_sor, itcount_sor] = it_sor(A, b, tol, sor_optimal_omega);

    % make sure we got roughly the same answer using every method
    assert(abs(x_cg-x_jacobi)<1e-5)
    assert(abs(x_cg-x_gs)<1e-5)
    assert(abs(x_cg-x_sor)<1e-5)

    fprintf('CG took            %3d iterations.\n', itcount_cg);
    fprintf('SOR took            %3d iterations.\n', itcount_sor);
    fprintf('Gauss-Seidel took %3d iterations.\n', itcount_gs);
```

```

    fprintf('Jacobi took      %3d iterations.\n', itcount_jacobi);
end
% -----
function assert(p)
    not_p = ~p;
    if (sum(abs(not_p)) ~= 0)
        error('Assertion violated.')
    end
end
% -----
function [x, itcount] = it_cg(A, b, tol)
    [n,dummy] = size(A);
    x = zeros(n,1);

    r = A*x - b;
    d = -r;
    delta = r'*r;

    itcount = 0;

    while 1
        u = A*d; % this is where we spend most of our processing time
        itcount = itcount + 1; % that's why we increment the iteration count here

        lambda = delta/(d'*u);
        x = x + lambda*d;
        r = r + lambda*u;
        next_delta = r'*r;

        if (sqrt(next_delta)<tol)
            return
        end

        alpha = next_delta/delta;
        delta = next_delta;
        d = -r + alpha*d;
    end
end
% -----
function [x, itcount] = it_jacobi(A, b, tol)
    [dummy, n] = size(A);

    L = -tril(A, -1);
    U = -triu(A, 1);
    D = diag(diag(A));

    T = D\(L+U);
    c = D\b;

    [x, itcount] = run_iteration(T, c, zeros(n,1), tol);
end
% -----
function [x, itcount] = it_gauss_seidel(A, b, tol)
    [dummy, n] = size(A);

```

```

L = -tril(A, -1);
U = -triu(A, 1);
D = diag(diag(A));

T = (D-L)\U;
c = (D-L)\b;

[x, itcount] = run_iteration(T, c, zeros(n,1), tol);
end
% -----
function [x, itcount] = it_sor(A, b, tol, omega)
[dummy, n] = size(A);

L = -tril(A, -1);
U = -triu(A, 1);
D = diag(diag(A));

T = (1/omega*D-L)\((1/omega-1)*D+U);
c = (1/omega*D-L)\b;

[x, itcount] = run_iteration(T, c, zeros(n,1), tol);
end
% -----
function [x, itcount] = run_iteration(T, c, xstart, tol)
x = xstart;
last_stepsize = 2*tol;

itcount = 0;

while last_stepsize > tol
    last_x = x;
    x = T*x + c;
    last_stepsize = norm(x-last_x, 2);
    itcount = itcount + 1;
end
end

```

2. Problem 13, p. 249

We choose to determine a_0, \dots, a_3 by means of the conjugate gradient method. In order to be able to do that, we confirm that the matrix (which is symmetric by inspection) is positive definite. One way to perform this verification is by Choleksy decomposition, which we can simply achieve numerically.

We obtain the following results:

```

A =
    8.0000    14.0000    35.0000    98.0000
   14.0000    35.0000    98.0000   292.2500
   35.0000    98.0000   292.2500   906.5000
   98.0000   292.2500   906.5000  2887.8125

```

```

b =

```

```
13.300
25.450
67.625
202.637
```

```
x_cg =
```

```
0.88030
3.39048
-2.72165
0.55152
```

```
x_matlab_backslash =
```

```
0.88030
3.39048
-2.72165
0.55152
```

We thus obtain the polynomial

$$0.88030 + 3.39048x - 2.72165x^2 + 0.55152x^3$$

as the least-squares fit through these data.

The code for this problem is the following:

```
function p3
    A = [8,14,35,98; ...
         14,35,98,292.25; ...
         35,98,292.25,906.5; ...
         98,292.25,906.5,2887.8125]

    b = [13.3;25.45;67.625;202.6375]

    % Verify positive definiteness:
    L = chol(A);
    assert(abs(L'*L-A)<1e-12);

    x_cg = solve_by_cg(A, b, 1e-15)
    x_matlab_backslash = A \ b
end
function assert(p)
    not_p = ~p;
    if (sum(abs(not_p)) ~= 0)
        error('Assertion violated.')
    end
end
function x = solve_by_cg(A, b, tol)
    [n,dummy] = size(A);
    x = zeros(n,1);
    r = A*x - b;
```

```
d = -r;
delta = r'*r;
while 1
    u = A*d;
    lambda = delta/(d'*u);
    x = x + lambda*d;
    r = r + lambda*u;
    next_delta = r'*r;

    if (sqrt(next_delta)<tol)
        return
    end

    alpha = next_delta/delta;
    delta = next_delta;
    d = -r + alpha*d;
end
end
```