```
% set global config options
global iters;
global tol;
iters = 2000;
tol = 10^-6;

% setup plot
f = figure();
title('2.b Convergence of methods');
ploti = 0;

for N = [31 63]
  for alpha = sqrt([0 10 1000])
    % increment plot number
    ploti = ploti + 1;

    % compute B and b from Bx=b
    n = N^2;
    h = 1/(N+1);
    c = (alpha * h)^2;
    B = delsq(numgrid('S', N + 2)) + c*speye(n);
    b = B*ones(N^2,1);

    % x is the initial guess
    x = zeros(n, 1);

    k2 = (4 -2 * (cos(floor(N) * pi /(N+1)) + cos(floor(N) * pi / (N+1))) + (alpha./
(N+1)).^2)./(4-2*(cos(pi./(N+1)) + cos(pi./(N+1))) + (alpha./(N+1)).^2);
    fprintf('N = %d, n = %d, k2(B) = %d\n', N, n, k2);

    % compute the residuals and iteration counts for each method.
    [~, jr, i] = jacobi(B, x, b);
    fprintf('jacobi: %d iters\n', i)
    [~, gsr, i] = gaussSeidel(B, x, b);
    fprintf('gaussSeidel: %d iters\n', i)
    [~, sorr, i] = SOR(B, x, b);
    fprintf('SOR: %d iters\n', i)
    [~, cgr, i] = CG(B, x, b);
    fprintf('cg: %d iters\n', i)
    [~, pcgr, i] = PCG(B, x, b);
    fprintf('pcg: %d iters\n', i)

    % plot the residual vectors
    subplot(3, 2, ploti);
    semilogy(getx(jr), jr, getx(gsr), gsr, getx(sorr), sorr, getx(cgr), cgr, getx(pc
gr), pcgr);
    legend('jacobi', 'gaussSeidel', 'SOR', 'CG', 'PCG');
    xlabel('iterations');
    ylabel('relative residual');
    title(sprintf('N = %d, alpha = %d', N, alpha))

    fprintf('\n');
  end
end

% save plot
saveas(f, 'q2.png');

%% getx returns a vector 1, 2, ..., len(b).
function x = getx(b)
  x = 1:max(size(b));
end

%% gaussSeidel solves Bx=b using the Gauss-Seidel method.
function [x, relres, i]  = gaussSeidel(B, x, b)
  global iters;
  global tol;

  % Create matrix splitting of B
  D = diag(diag(B));
  L = -1*tril(B,-1);
  U = -1*triu(B,1);
```

```
  % Compute iteration matrix
  M = inv(D-L);

  r0 = residual(B, x, b);
  rs = [r0];
  for i = 1:iters
    x = M * (b + U * x);  % compute current solution
    r = residual(B, x, b);
    rs = [rs, r];
    if r/r0 < tol
      break
    end
  end
  relres = rs/norm(b, 2);
end

%% jacobi solves Bx=b using the Jacobi method.
function [x, relres, i] = jacobi(B, x, b)
  global iters;
  global tol;

  % Create matrix splitting of B
  D = diag(diag(B));
  L = -1*tril(B,-1);
  U = -1*triu(B,1);

  % Compute iteration matrix
  M = inv(D);

  % cache LPU for better performance
  LPU = (L+U);

  r0 = residual(B, x, b);
  rs = [r0];
  for i = 1:iters
    x = M*(LPU*x + b);  % compute current solution
    r = residual(B, x, b);
    rs = [rs, r];
    if r/r0 < tol
      break
    end
  end
  relres = rs/norm(b, 2);
end

%% CG solves Bx=b using the Conjugate Gradient method.
function [x, relres, i] = CG(B, x, b)
  global iters;
  global tol;

  [x, ~, ~, i, relres] = pcg(B, b, tol, iters, [], [], x);
end

%% PCG solves Bx=b using the Conjugate Gradient method with a incomplete
%% Cholesky IC(0).
function [x, relres, i] = PCG(B, x, b)
  global iters;
  global tol;

  L = ichol(B);
  [x, ~, ~, i, relres] = pcg(B, b, tol, iters, L, L', x);
end

function [x, relres, i] = SOR(B, x, b)
  global iters;
  global tol;

  % Create matrix splitting of B
  D = diag(diag(B));
  L = -1*tril(B,-1);
  U = -1*triu(B,1);

  % Compute iteration matrix
```

```
  M = inv(D-L);
  jacobiM = inv(D);

  % compute damping factor
  spectral_radius = max(abs(eig(jacobiM)));
  w = 2 / (1 + sqrt(1 - spectral_radius^2));

  r0 = residual(B, x, b);
  rs = [r0];
  for i = 1:iters
    xnew = M * (b + U * x);  % compute GS
    x = w * xnew + (1-w) * x; % damp GS according to damping factor w
    r = residual(B, x, b);
    rs = [rs, r];
    if r/r0 < tol
      break
    end
  end
  relres = rs/norm(b, 2);
end

%% residual computes the residual of Bx = b
function r = residual(B, x, b)
  r = norm(b - B*x, 2);
end
```