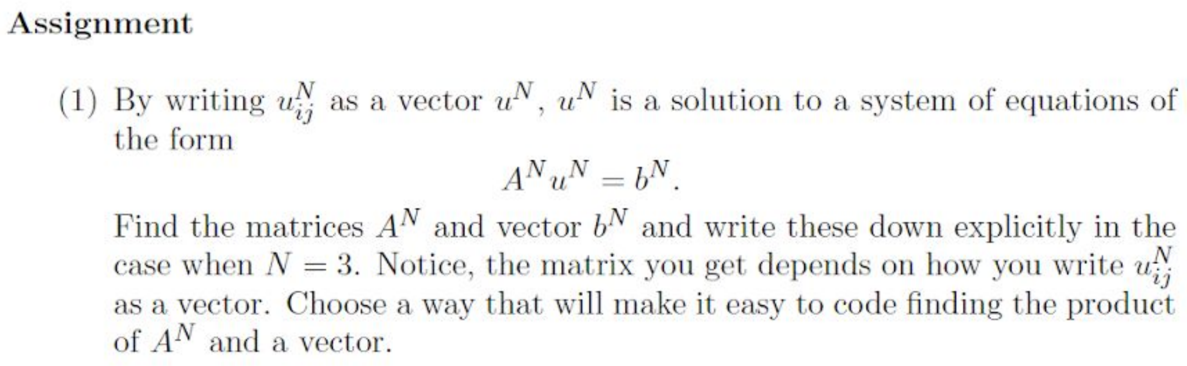
Project 1
Math 458
Kartik, Heather, Saket
Github Link to All Code: https://github.com/kartikbalodi/math458_project1



Assignment

(1) By writing u_{ij}^N as a vector u^N, u^N is a solution to a system of equations of the form

$$A^N u^N = b^N.$$

Find the matrices A^N and vector b^N and write these down explicitly in the case when $N = 3$. Notice, the matrix you get depends on how you write u_{ij}^N as a vector. Choose a way that will make it easy to code finding the product of A^N and a vector.

We write $u_{ij}{}^N$ as a N^2x1 vector, which corresponds to $1 \le i \le N, 1 \le j \le N$. Thus, the matrix A^N take the form of a N^2xN^2 array and b^N takes the form of a N^2x1 vector.

For $u^3 = [\ u_{1,1}\ \ u_{2,1}\ \ u_{3,1}\ \ u_{1,2}\ \ u_{2,2}\ \ u_{3,2}\ \ u_{1,3}\ \ u_{2,3}\ \ u_{3,3}\ ]^T$, we express b^3 and A^3

$$b^3 = [\ b_{1,1}\ \ b_{2,1}\ \ b_{3,1}\ \ b_{1,2}\ \ b_{2,2}\ \ b_{3,2}\ \ b_{1,3}\ \ b_{2,3}\ \ b_{3,3}\ ]^T = [\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ ]^T$$

$$A^3 = \begin{matrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{matrix}$$

Code:

```
N = 10; % N

% initialize b to g(xij,  yij)
h = 1/(N+1);
b = zeros(N^2,1);
for i = 1:N
      for j = 1:N
            if i/(N+1) > 1/5 && i/(N+1) < 3/5 && j/(N+1) > 1/4 && j/(N+1)<1/2
                  b((j-1)*N+i,1) = -h^2; % bij = -h^2*g(xij,yij
            end
      end
end

% filling in A
A = zeros(N^2,N^2);
for j = 1:N
      for i = 1:N
            %
            %    B I 0            4 1 0
            % A= I B I where B = 1 4 1 , I is 3x3 negative identity matrix and
            %    0 I B            0 1 4   0 is 3x3 zero matrix
            % we use this to get
            % 4ui;j - ui+1;j - ui-1;j - ui;j+1 - ui;j-1
            % fill in identity matrices on the side diagonals
            c = (j-1)*N;
            if j > 1
                  A(c+i-N, c+i) = -1;
            end
            if j < N
                  A(c+i+N, c+i) = -1;
            end
            % then fill in B
            A(c+i,c+i)= 4;
            if i ~= 1
                  A(c+i, c+i-1) = -1;
            end
            if i ~= N
                  A(c+i, c+i+1) = -1;
            End

      end
end

fmt = [repmat('%4d ', 1, size(A,2)-1), '%4d\n'];
fprintf(fmt, A.');   %transpose is important
```

(2) Write a matlab script that generates the matrix A^N and vector b^N for any N and solves the system of equations using each of the following methods.

- Jacobi method,
- Gauss-Seidel with successive over-relaxation with any parameter ω, and
- the conjugate gradient method.

Github: https://github.com/kartikbalodi/math458_project1

Jacobi:

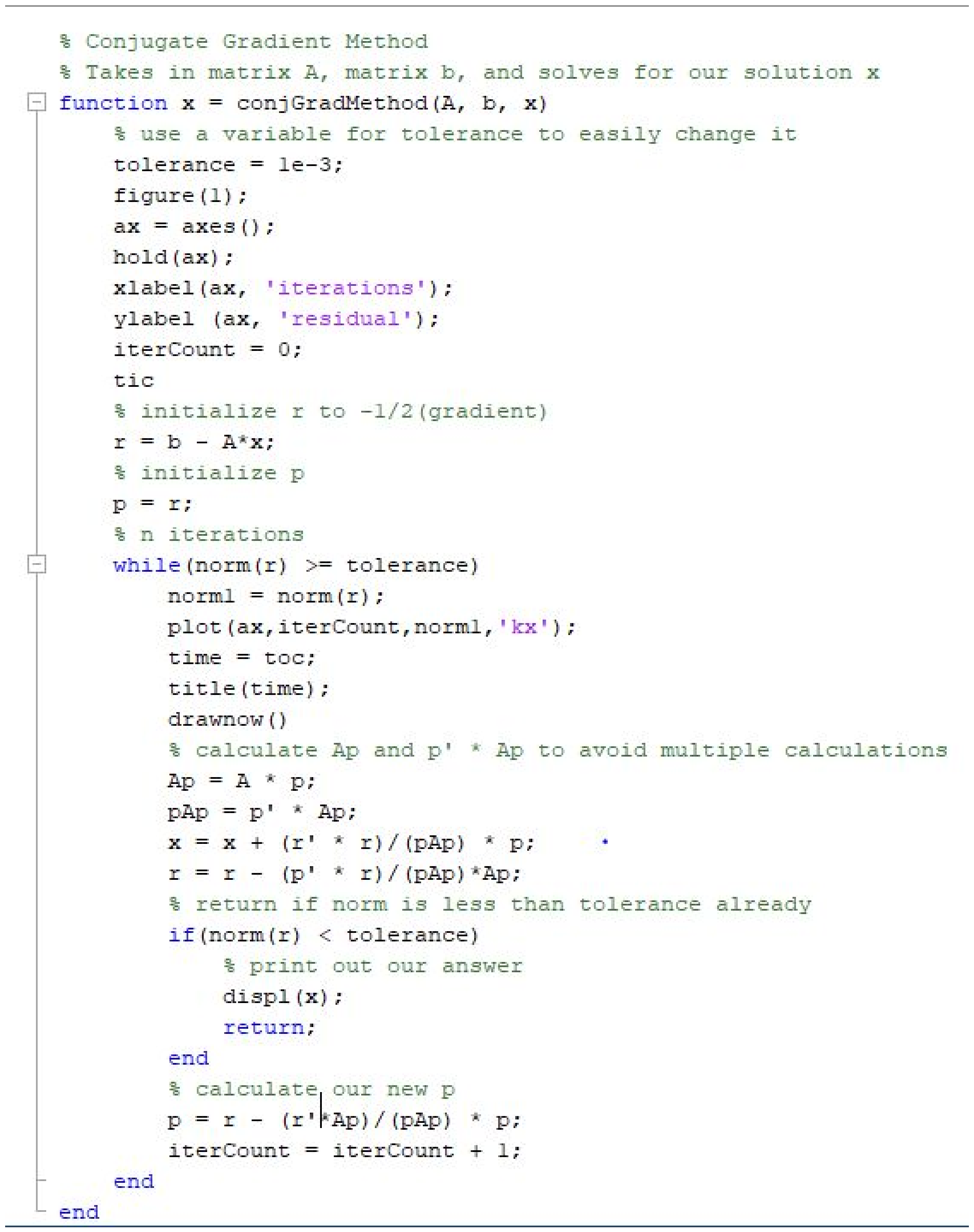
```
%--------------------------------------------------------------------------
% Start of Jacobi Method
%--------------------------------------------------------------------------

% these are initializations which are used for every program
atol = 1e-3; % absolute tolerance
programTimeLimit = 300; % runtime limit in seconds
x0 = zeros(N^2, 1); % initial guess
r = norm(b - A *x0, 1); % initial residual (b-A*xk)
xk = x0; % kth iteration
iterCount = 0; % number of iterations completed

figure(1);
ax = axes();
hold(ax)
xlabel(ax, 'iterations')
ylabel (ax, 'residual')
%title(ax,'residual against time/s and iteration/count')
ax_top = axes();
hold(ax_top)
ax_top.Position = ax.Position;
ax_top.YAxis.Visible = 'off';
ax_top.XAxisLocation = 'top';
xlabel(ax_top, 'time')

tic
% repeat iterations until r <= atol or program time limit reached
while r > atol
         plot(ax,iterCount,r,'kx');
         time = toc;
         plot(ax_top,time,r,'kx');
         if time > programTimeLimit
                  disp('Program time limit '+programTimeLimit+' reached');
         break
         end
         drawnow

         % find the x in next iteration by given formula
         iterNext = zeros(N^2,1);
         for i = 1:N^2
                  sum = 0;
                  for j = 1:N^2
                           if j ~= i
                                    sum = sum + A(i,j)* xk(j, 1);
                           end
                  end
                  iterNext(i, 1) = (b(i,1)-sum)/A(i,i);
         end
         xk = iterNext;
         r = norm(b - A *xk, 1); % update residual
         iterCount = iterCount + 1; % increment iteration count
end
% plot one more time for the last pt i.e. first r > atol
plot(ax,iterCount,r,'kx');
time = toc;
plot(ax_top,time,r,'kx');
drawnow

disp('N = ');
disp(N);
disp('time = ');
disp(time);
disp('iterations = ');
disp(iterCount);


%hold(ax_top,'off')
%hold(ax,'off')
```

Gauss-Seidel with SOR:

```
%Stopping Conditions: a tolerance limit and max number of
iterations
%before quitting
tolerance = 1e-12;
max_iterations = 100000;
%LDU Decomposition of A
%Isolates lower diagonal (below the main diagonal)
L = -tril(A, -1);
%Isolates main diagonal
D = diag(diag(A));
%Isolates upper diagonal (above the main diagonal)
U = -triu(A, 1);
%Creates initial solution array of zeros
x = zeros(length(A));
%Stored in reference since it is used repeatedly
%A in the equation Ax_i = b where x_i represents the current
%estimation of x
ref = (D - omega * L);
%Repeats until stopping criterion of iterations is met
for i = 1:max_iterations
     %Solving the iteration step
     x =
mldivide(ref,((1-omega)*D+omega*U)*x_init)+omega*mldivide(ref
,b);
     %Ends the loop if stopping criterion of tolerance is
     %met
     %Tolerance is the max. acceptable distance between x
     %and x_init
     %(previous estimate of x)
     if norm(x - x_init) < tolerance
     break;
     end
     %x_init stores previous value of x
     x_init = x;
end
```

Conjugate Gradient:

%---
% Start of Conjugate Gradient Method
%---



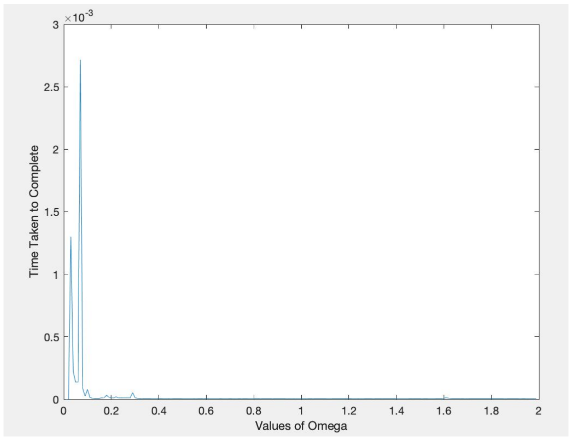
```
% Conjugate Gradient Method
% Takes in matrix A, matrix b, and solves for our solution x
function x = conjGradMethod(A, b, x)
    % use a variable for tolerance to easily change it
    tolerance = 1e-3;
    figure(1);
    ax = axes();
    hold(ax);
    xlabel(ax, 'iterations');
    ylabel (ax, 'residual');
    iterCount = 0;
    tic
    % initialize r to -1/2(gradient)
    r = b - A*x;
    % initialize p
    p = r;
    % n iterations
    while(norm(r) >= tolerance)
        norm1 = norm(r);
        plot(ax,iterCount,norm1,'kx');
        time = toc;
        title(time);
        drawnow()
        % calculate Ap and p' * Ap to avoid multiple calculations
        Ap = A * p;
        pAp = p' * Ap;
        x = x + (r' * r)/(pAp) * p;
        r = r - (p' * r)/(pAp)*Ap;
        % return if norm is less than tolerance already
        if(norm(r) < tolerance)
            % print out our answer
            displ(x);
            return;
        end
        % calculate our new p
        p = r - (r'*Ap)/(pAp) * p;
        iterCount = iterCount + 1;
    end
end
```

(3) Experiment with different values of the relaxation parameter ω to find an optimal choice for the SOR method. Describe how you did this and how you decided what choice was optimal.

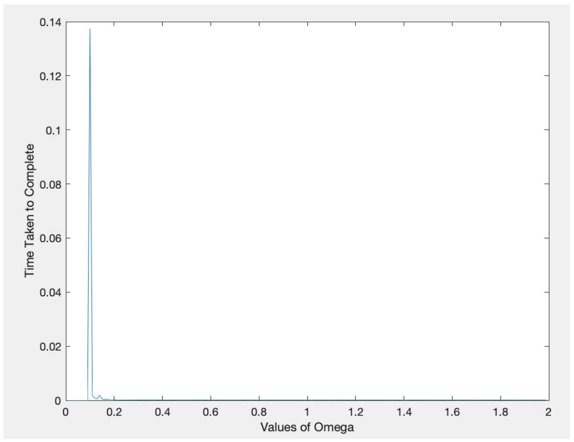
```
%Set value of omega, can change for tests
omega = 0.01:0.01:1.99;
x = zeros(N^2, 1);

least_time = 10000000;
best_omega = 0;
for i = i:length(omega)
    tic;
    x = SORMethod(A,b,x, omega(i));
    time = toc;
    if time < least_time
        best_omega = omega(i);
        least_time = time;
    end
end

function x = SORMethod(A,b,x_init, omega)
    %Stopping Conditions: a tolerance limit and max number of iterations
    %before quitting
    tolerance = 1e-12;
    max_iterations = 100000;

    %LDU Decomposition of A
    %Isolates lower diagonal (below the main diagonal)
    L = -tril(A, -1);
    %Isolates main diagonal
    D = diag(diag(A));
    %Isolates upper diagonal (above the main diagonal)
    U = -triu(A, 1);

    %Creates initial solution array of zeros
    x = zeros(length(A));

    %Stored in reference since it is used repeatedly
    %A in the equation Ax_i = b where x_i represents the current
    %estimation of x
    ref = (D - omega * L);

    %Repeats until stopping criterion of iterations is met
    for i = 1:max_iterations
        %Solving the iteration step
        x = mldivide(ref,((1-omega)*D+omega*U)*x_init)+omega*mldivide(ref,b);

        %Ends the loop if stopping criterion of tolerance is met
        %Tolerance is the max. acceptable distance between x and x_init
        %(previous estimate of x)
        if norm(x - x_init) < tolerance
            break;
        end
        %x_init stores previous value of x
        x_init = x;
    end
end
```

Above is the code used to calculate the optimal omega value for various values of N (generating the matrix from part (1). We found that the optimal omega values increase w.r.t. N, as shown in the following graphs:

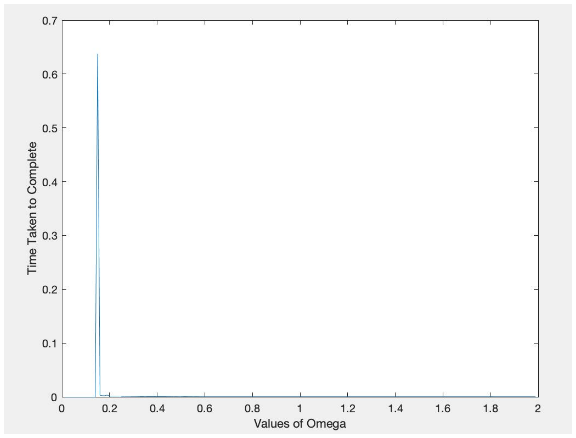
N = 3 (Best Omega = 0.990)



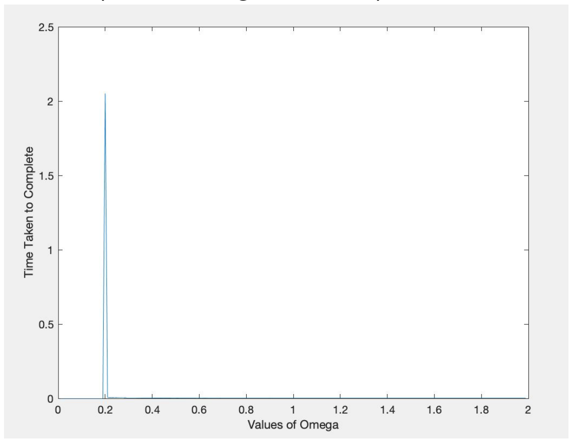
N = 10 (Best Omega = 1.410)



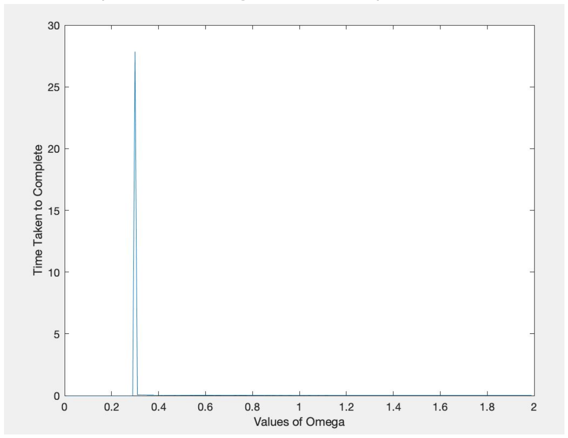
N = 15 (Best Omega = 1.680)



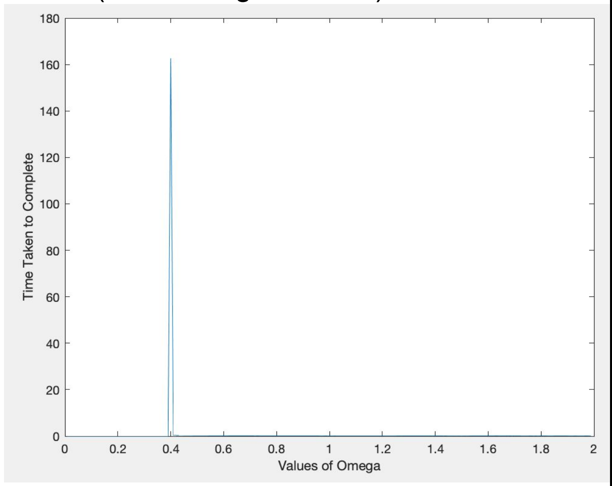
N = 20 (Best Omega = 1.830)

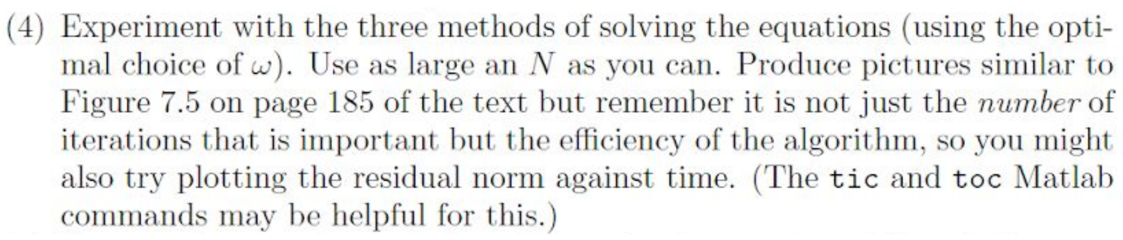


N = 30 (Best Omega = 1.930)



N = 40 (Best Omega = 1.940)





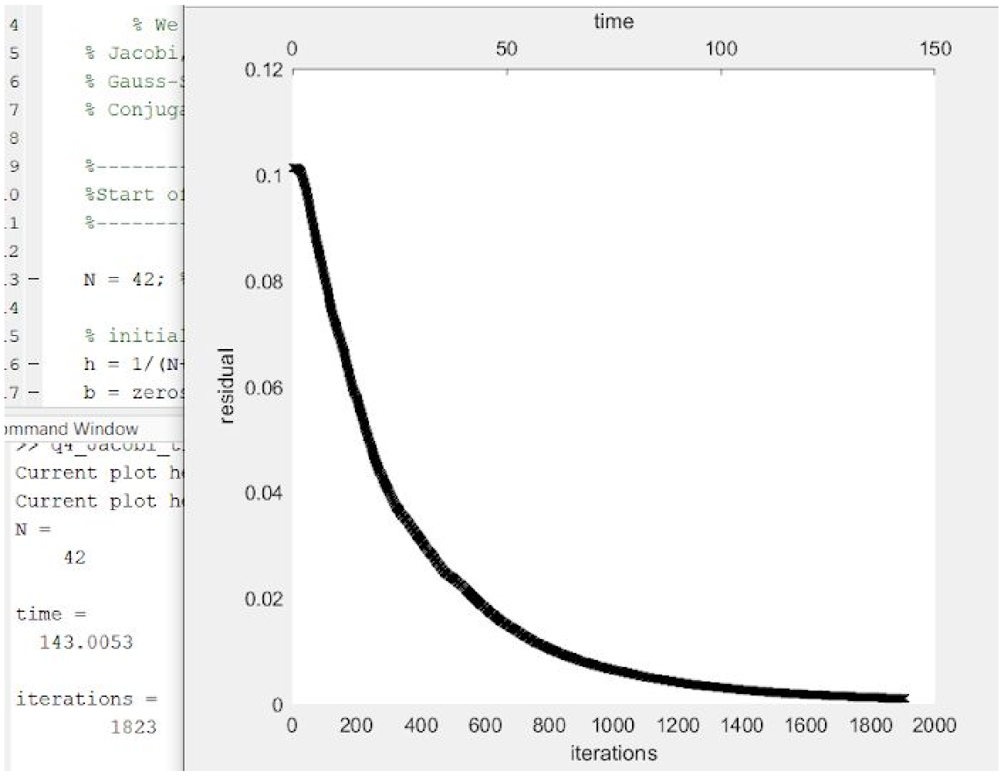
(4) Experiment with the three methods of solving the equations (using the optimal choice of ω). Use as large an N as you can. Produce pictures similar to Figure 7.5 on page 185 of the text but remember it is not just the *number* of iterations that is important but the efficiency of the algorithm, so you might also try plotting the residual norm against time. (The `tic` and `toc` Matlab commands may be helpful for this.)

Absolute tolerance was set to 1e-3, and N was measured and tabulated till time taken exceeded 150s.
All plots were plotted as residuals against time(top x-axis) and iteration count (bottom x-axis).

Jacobi Analysis:

N	Time Taken	Iteration Count
10	3.4230 seconds	118
20	11.6114 seconds	427
30	36.8058 seconds	946
40	105.5501 seconds	1643
42	**143.0053 seconds**	**1823**
43	166.9965 seconds	1879

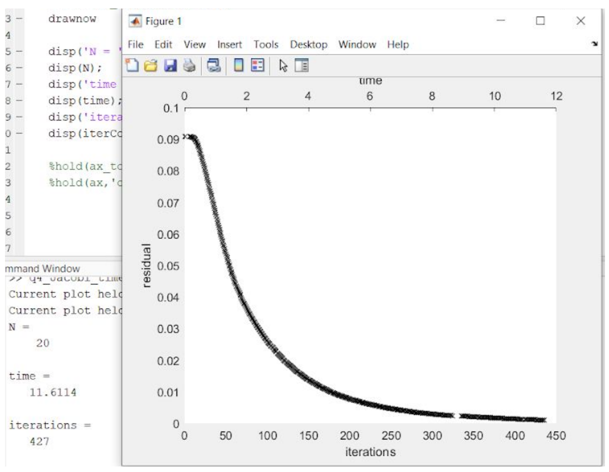
Highest N possible for Jacobi method is N=42.
Plot:



```
     % We
  % Jacobi,
  % Gauss-
  % Conjug

  %-------
  %Start o
  %-------

  N = 42;

  % initia
  h = 1/(N
  b = zero
```

```
ommand Window
>> q4_Jacobi_
Current plot h
Current plot h
N =

    42


time =

  143.0053


iterations =

        1823
```

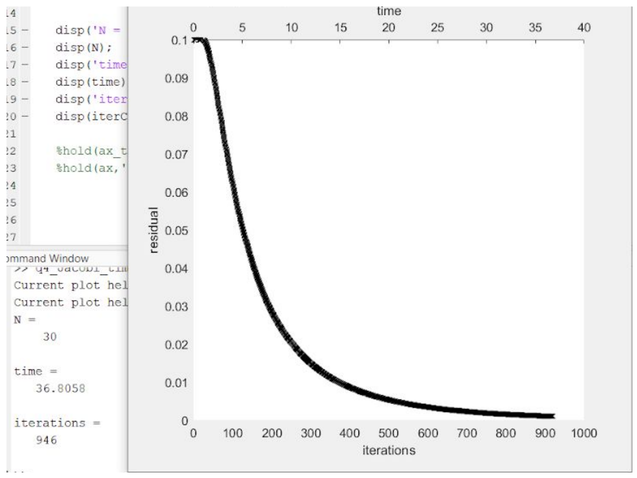
Plot for N=20, Jacobi:



```
    drawnow

    disp('N = '
    disp(N);
    disp('time
    disp(time);
    disp('itera
    disp(iterCo

    %hold(ax_to
    %hold(ax,'c
```

```
Current plot helc
Current plot helc
N =
    20

time =
   11.6114

iterations =
   427
```

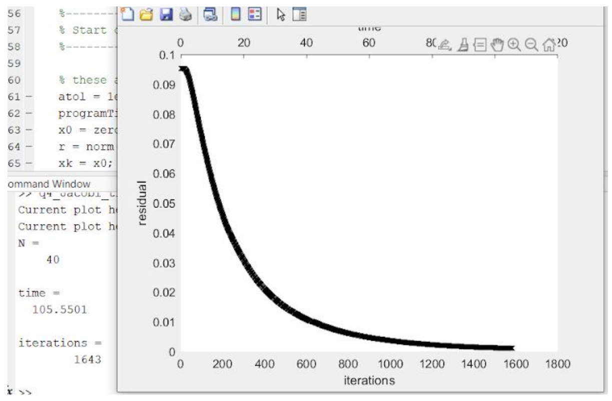
Plot for N=30, Jacobi:



```
    disp('N =
    disp(N);
    disp('time
    disp(time)
    disp('iter
    disp(iterC

    %hold(ax_t
    %hold(ax,'
```

```
Current plot hel
Current plot hel
N =
    30

time =
   36.8058

iterations =
   946
```

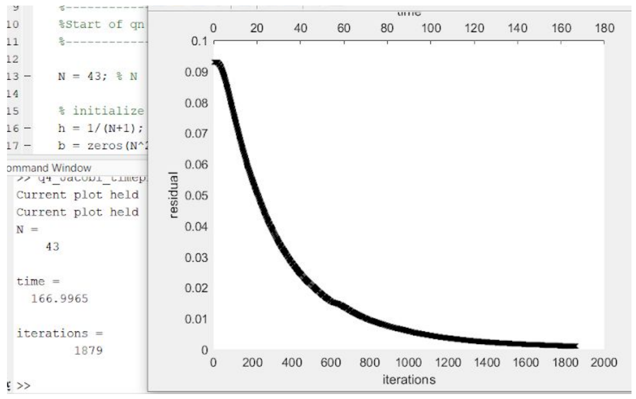
Plot for N=40, Jacobi:



```
    %-------
    % Start c
    %-------

    % these a
    atol = 1e
    programTi
    x0 = zero
    r = norm
    xk = x0;
```

```
Current plot hc
Current plot hc
N =
    40

time =
  105.5501

iterations =
        1643

>>
```

Plot for N=43, Jacobi:



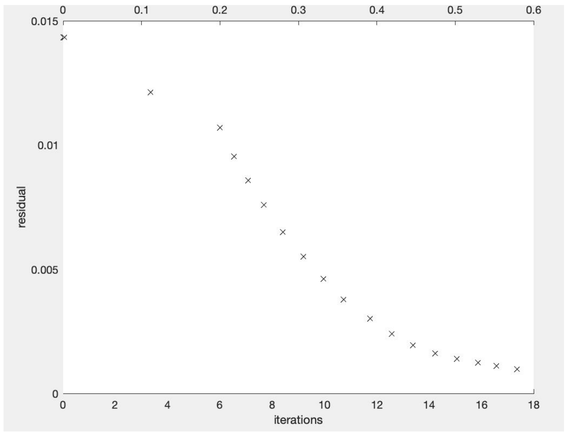
```
    %-----------
    %Start of qn
    %-----------

    N = 43; % N

    % initialize
    h = 1/(N+1);
    b = zeros(N^
```

```
Current plot held
Current plot held
N =
    43

time =
  166.9965

iterations =
        1879

>>
```

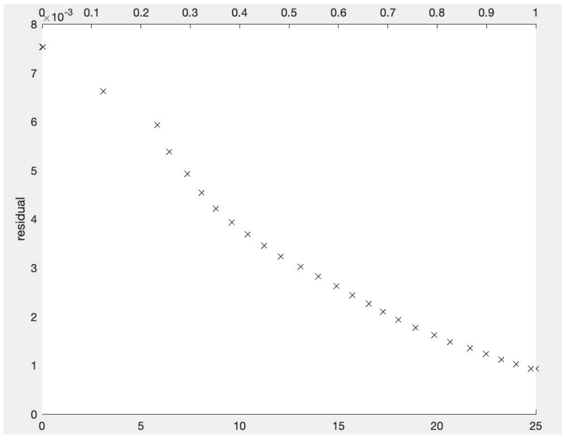
Gauss-Seidel:

N	Time Taken	Iteration Count
20	0.579 seconds	18
40	0.9904 seconds	25
60	3.038 seconds	31
100	15.83 seconds	35
125	**69.464 seconds**	**35**
130	205.894 seconds	36

Plot for N = 20:

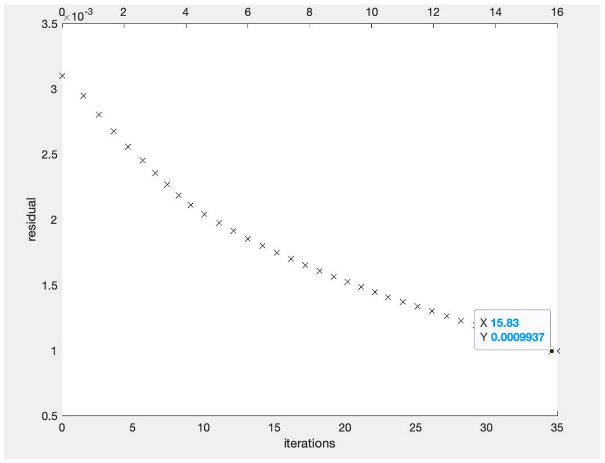


Plot for N = 40:

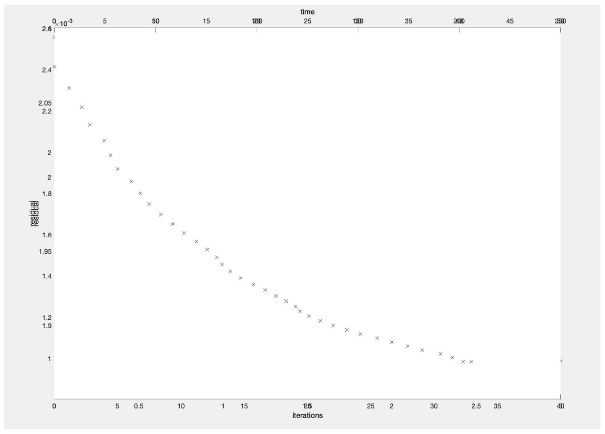


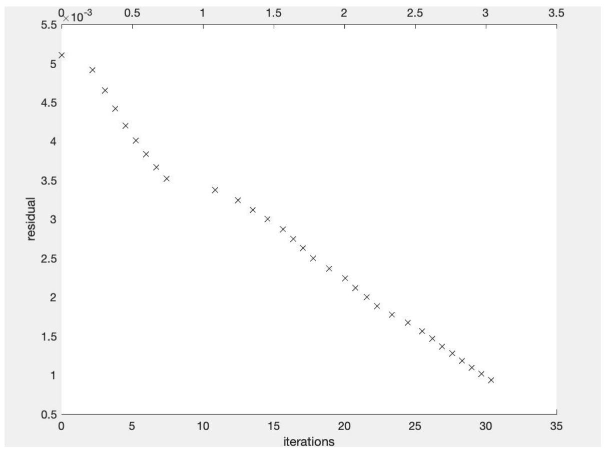
Plot for N = 60:

Plot for N = 100:

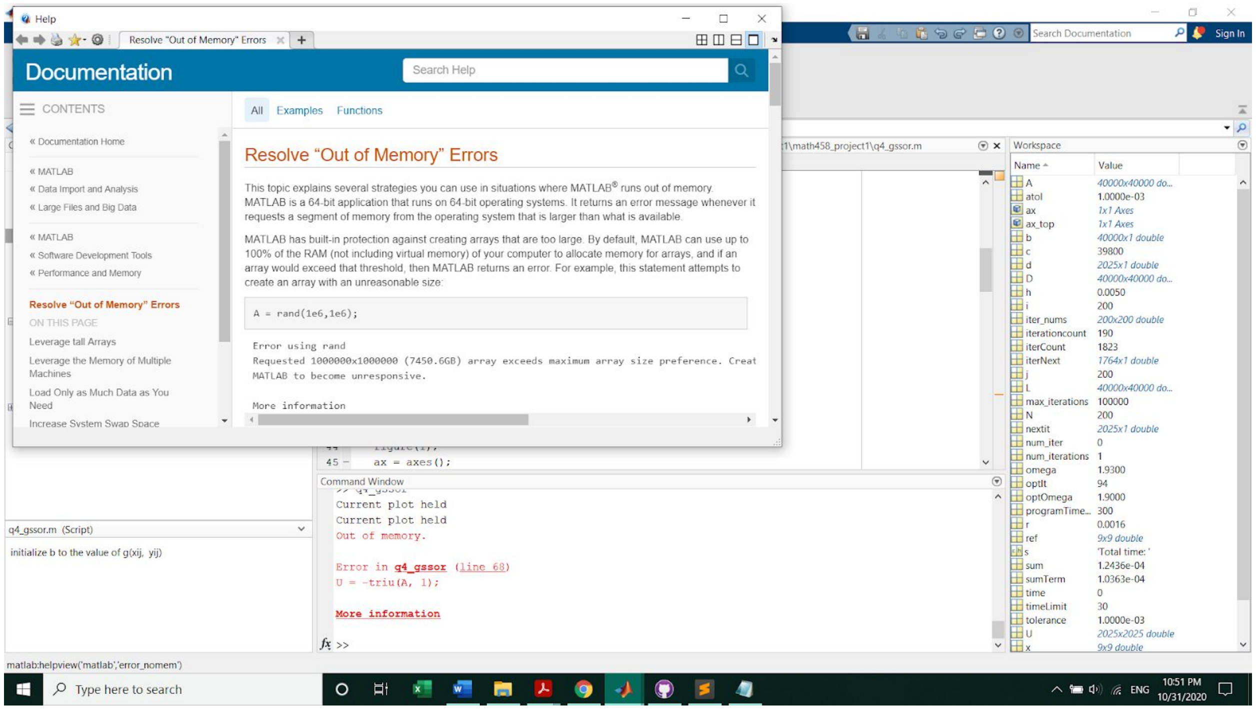


Plot for N = 130





At 200, Matlab runs out of memory:


Help
Resolve "Out of Memory" Errors
Documentation
Search Help
CONTENTS
All
Examples
Functions
« Documentation Home
« MATLAB
« Data Import and Analysis
« Large Files and Big Data
« MATLAB
« Software Development Tools
« Performance and Memory
Resolve "Out of Memory" Errors
ON THIS PAGE
Leverage tall Arrays
Leverage the Memory of Multiple Machines
Load Only as Much Data as You Need
Increase System Swap Space
Search Documentation
Sign In
Command Window
q4_gssor.m (Script)
matlab:helpview('matlab','error_nomem')
Type here to search
ENG
10:51 PM
10/31/2020

Resolve "Out of Memory" Errors

This topic explains several strategies you can use in situations where MATLAB® runs out of memory. MATLAB is a 64-bit application that runs on 64-bit operating systems. It returns an error message whenever it requests a segment of memory from the operating system that is larger than what is available.

MATLAB has built-in protection against creating arrays that are too large. By default, MATLAB can use up to 100% of the RAM (not including virtual memory) of your computer to allocate memory for arrays, and if an array would exceed that threshold, then MATLAB returns an error. For example, this statement attempts to create an array with an unreasonable size:

```
A = rand(1e6,1e6);
```

```
Error using rand
Requested 1000000x1000000 (7450.6GB) array exceeds maximum array size preference. Creat
MATLAB to become unresponsive.

More information
```

```
ax = axes();
```

initialize b to the value of g(xij, yij)

```
Current plot held
Current plot held
Out of memory.

Error in q4_gssor (line 68)
U = -triu(A, 1);

More information
>>
```

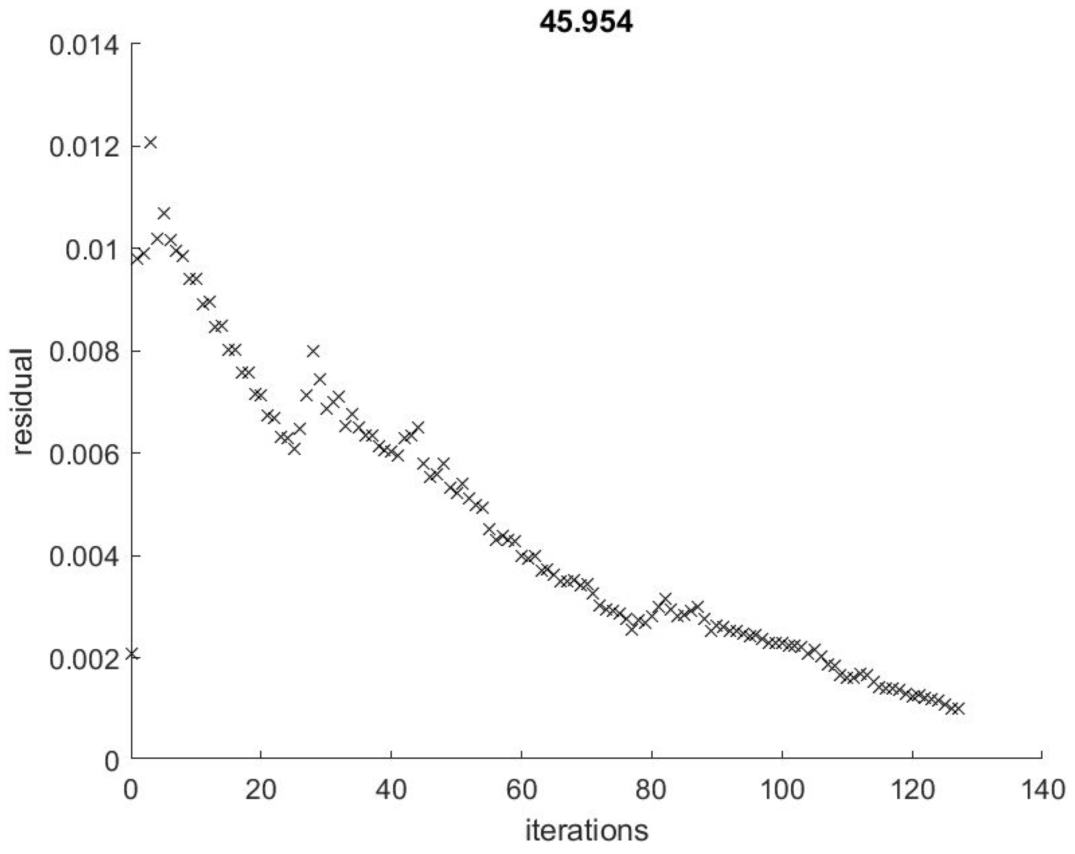
Workspace

Name	Value
A	40000x40000 do...
atol	1.0000e-03
ax	1x1 Axes
ax_top	1x1 Axes
b	40000x1 double
c	39800
d	2025x1 double
D	40000x40000 do...
h	0.0050
i	200
iter_nums	200x200 double
iterationcount	190
iterCount	1823
iterNext	1764x1 double
j	200
L	40000x40000 do...
max_iterations	100000
N	200
nextit	2025x1 double
num_iter	0
num_iterations	1
omega	1.9300
optIt	94
optOmega	1.9000
programTime...	300
r	0.0016
ref	9x9 double
s	'Total time: '
sum	1.2436e-04
sumTerm	1.0363e-04
time	0
timeLimit	30
tolerance	1.0000e-03
U	2025x2025 double
x	9x9 double

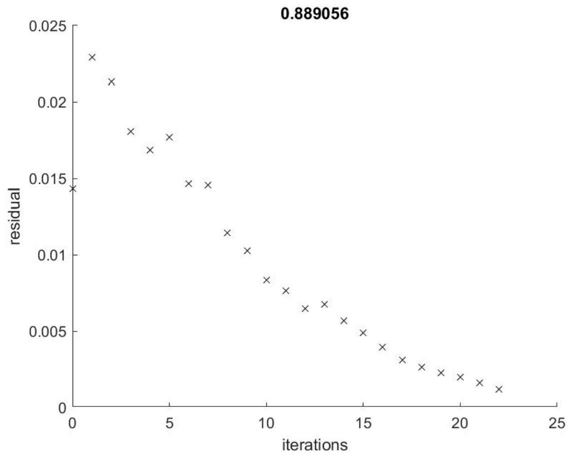
Conjugate Gradient Analysis:

N	Time Taken	Iteration Count
20	.889 seconds	22
50	2.284 seconds	50
100	6.208 seconds	93
120	12.002 seconds	110
130	15.493 seconds	114
145	34.872 seconds	126
149	**45.954 seconds**	**127**
150	350.591 seconds	131

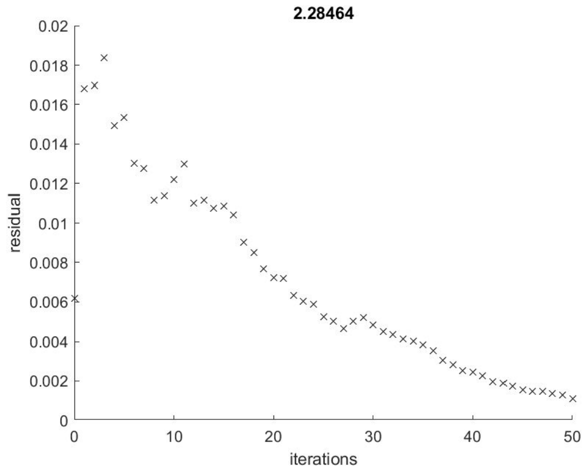
Plot for winning time (N = 149):



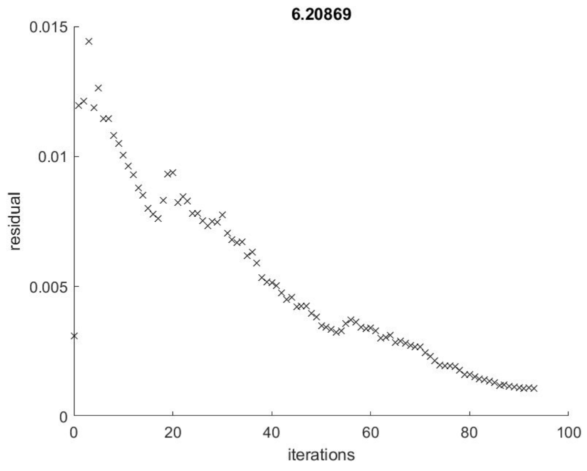
Plot for N = 20 (CGM):



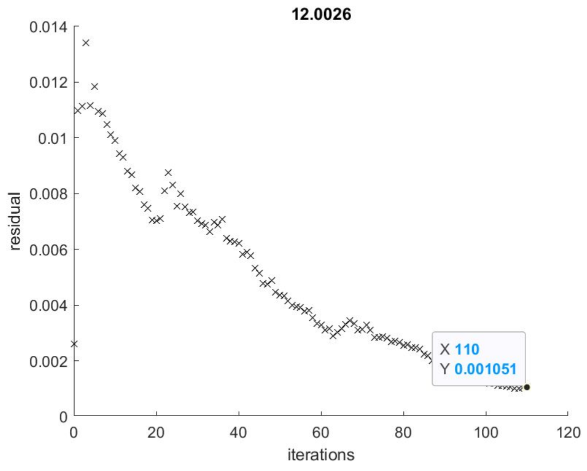
Plot for N = 50 (CGM):



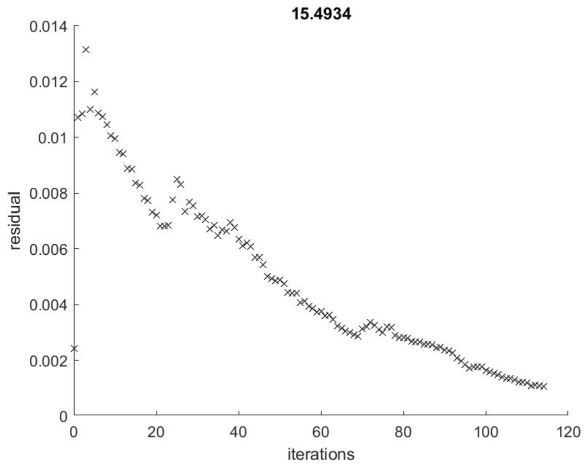
Plot for N = 100 (CGM):



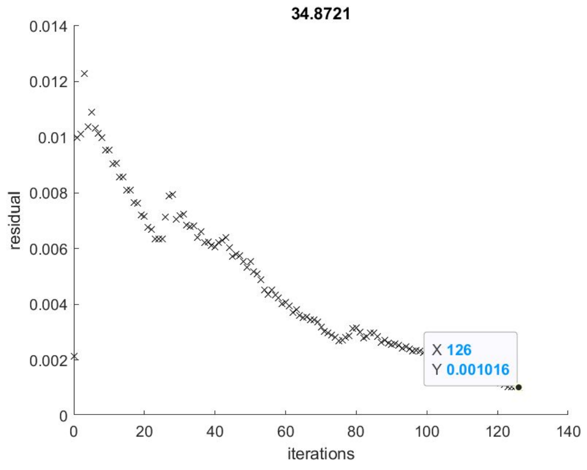
Plot for N = 120 (CGM):



Plot for N = 130 (CGM):



Plot for N = 145 (CGM):



(5) Use your best performing algorithm with a large value of N and plot your solution.

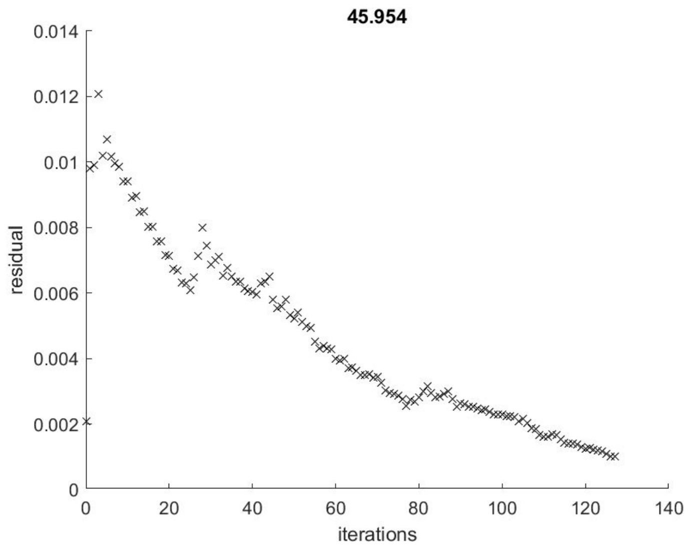
Conjugate gradient method is far more superior in comparison to Gauss-Seidel with SOR, which in turn is far better than Jacobi.

Something interesting to note is the large change in time between N=149 and N=150, despite the iterations being approximately equal. Our theory is that matlab may have ran out of the space on the stack needed to perform the computations, which would require lengthy reshuffling of space. This would significantly increase the time of the computations, and would help explain the steep incline of time taken. This is further corroborated by the fact that each of the consecutive N values (151, 152, …) were also consistent with the large jump in time taken.

If this is the case, our maximum N value could have been notably larger, judging from the relatively short time that N=149 took.

Our plot is below. As you can see, the number of iterations (127) was relatively small compared to the other two methods for their maxed out N.

Conjugate Gradient Method Solution:
Plot for winning time (N = 149):



(6) For each method, tell me who it was in your group who took primary responsibility for coding it up. Also, briefly describe the contributions of each of you to the rest of the project.

Kartik:
Code for part (1), Jacobi + plots & time data for Jacobi. Corresponding sections of writeup.
Saket:
Code for part (3), Gauss-Seidel + plots & time data for Gauss-Seidel. Corresponding sections of writeup.
Heather:
Code for conjugate gradient + plots & time data for conjugate gradient. Corresponding sections of writeup, part(5).