21-341 Final Report: SOR, SSOR, USSOR

Huayun Huang

May 12, 2019

1. Background

1.1. SOR

Given a problem $Ax = b$, the successive over-relaxation method (SOR) is an iterative method that uses a relaxation factor that takes the strictly lower part of A and factors it into part of the iteration matrix. Its formula is given by:

$$(D + \omega L)x^{(k+1)} = [(1 - \omega)D - \omega U]x^{(k)} + \omega b$$

where D is the diagonal of A, L is the strictly lower triangle matrix of A, and U is the strictly upper triangle matrix of A. Here ω denotes the relaxation factor. When $\omega = 1$, the SOR is the same as the Gauss-Seidel method. The problem is solved with triangular solver during each iteration (class notes).

1.2. Exploring beyond SOR

The author of this paper would like to examine further beyond the other iterative methods with a relaxation parameter besides SOR. Throughout the research, the author came across Symmetric Successive Over-relaxation (SSOR), which was briefly mentioned during the class, and Unsymmetric Successive Over-relaxation (USSOR). Throughout this report we will use the notation style from the class.

2. SSOR

The Symmetric Successive Over-relaxation, also known as SSOR, contains two "sweeps" during each iteration. It was first proposed by Sheldon in 1955. (Sheldon, 1955)

2.1. Iterations

For each iteration, the SSOR method first performs a forward SOR to optimize $x^{(k)}$ to $x^{(k+\frac{1}{2})}$, and then performs a backward SOR to optimize $x^{(k+\frac{1}{2})}$ to $x^{(k+1)}$.

Step 1 (forward SOR):

$$(D + \omega L)x^{(k+\frac{1}{2})} = (-\omega U + (1 - \omega)D)x^{(k)} + \omega b$$

Step 2 (backward SOR):

$$(D + \omega U)x^{(k+1)} = (-\omega L + (1 - \omega)D)x^{(k+\frac{1}{2})} + \omega b$$

During each step, we can use a triangular solver to solve for $x^{(k+\frac{1}{2})}$ on step 1, and $x^{(k+1)}$ on step 2. Its iteration matrix thus becomes:

$$M_{SSOR} = (D+\omega U)^{-1}[-\omega L + (1-\omega)D](D+\omega L)^{-1}[-\omega U + (1-\omega)D]$$

2.2. Performance

The author coded the method in Python using the Numpy package (see appendix for the source code). For the problem $Ax = b$, the author uses the guideline from homework 5, where the SPD matrix A is generated using $n = 3, 4, ..., 10$ (so A has dimensions of $9, 16, ..., 100$, respectively). The real solution is set to be $b = A[1\ 1\ ...\ 1]^T$, and the initial guess $x^0 = [0\ 0\ 0\ ...\ 0]^T$. The tolerance is 10^{-8}, calculated by $\frac{||b-Ax||_2}{||b||_2}$, with a maximum allowed iteration of 10^4.

Figure 1 shows a chart of $\omega \in [0.1, 1.9]$ plotted against the number of iterations. The red curve in the plot denotes the optimal value of ω. Similar to the result obtained in homework 5, the optimal ω is rising as n increases. Note that, since SSOR does two sweeps of SOR, when comparing the two algorithms, we need to multiply the number of iterations from SSOR by 2.

We obtained a similar trending about the algorithm's run time (Figure 2): both ω and the run time go up as n increases.

2.3. Converge

Similar to the SOR method, SSOR conveges when A is SPD and $\omega \in (0, 2)$. When ω is optimized such that $\rho(M_{SSOR})$ is minimized, we obtain the fatest convergence rate (Evans & Forrignton, 1963).

3. USSOR

The author moved on to explore the Unsymmetric Successive Over-relaxation method (USSOR), which is very similar to the SSOR method but with two different relaxation factor on each sweep.

3.1. Iterations

The iterations in USSOR is very similar to the one descibed in section 2.1, except that for the forward sweep, we replace ω with a different relaxation factor, σ. (Young, 1964; D'Sylva & Miles, 1964)

Step 1 (forward SOR):

$$(D+\sigma L)x^{(k+\frac{1}{2})} = (-\sigma U + (1-\sigma)D)x^{(k)} + \sigma b$$

Step 2 (backward SOR):

$$(D+\omega U)x^{(k+1)} = (-\omega L + (1-\omega)D)x^{(k+\frac{1}{2})} + \omega b$$

Like SSOR, we also apply the triangular solver in each step. Its iteration matrix is:

$$M_{USSOR} = (D+\omega U)^{-1}[-\omega L + (1-\omega)D](D+\sigma L)^{-1}[-\sigma U + (1-\sigma)D]$$

When $\sigma = \omega$, USSOR is equivalent to SSOR.

3.2. Performance

The author coded the USSOR method with similar parameters as stated in section 2.2. However, since we have two relaxation parameters, we need to visalize their performance in a different way than what was presented in Figure 1 and Figure 2.

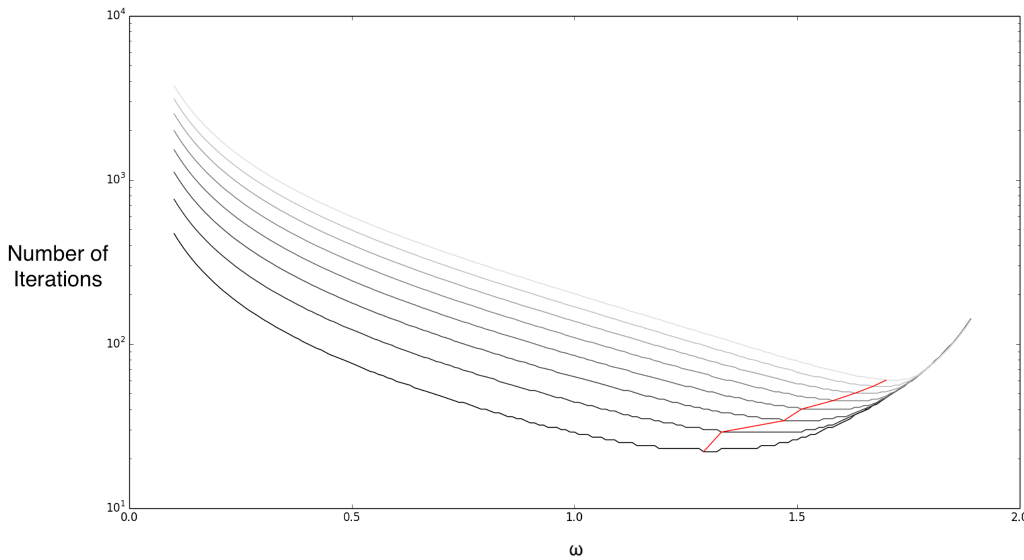

Fig. 1: Number of Iterations using the matrices from Homework 5. The darkest curve represents $n = 3$, while the lightest curve represents $n = 10$. The red curve denotes the optimal ω, meaning that there are least number of iterations under that ω.

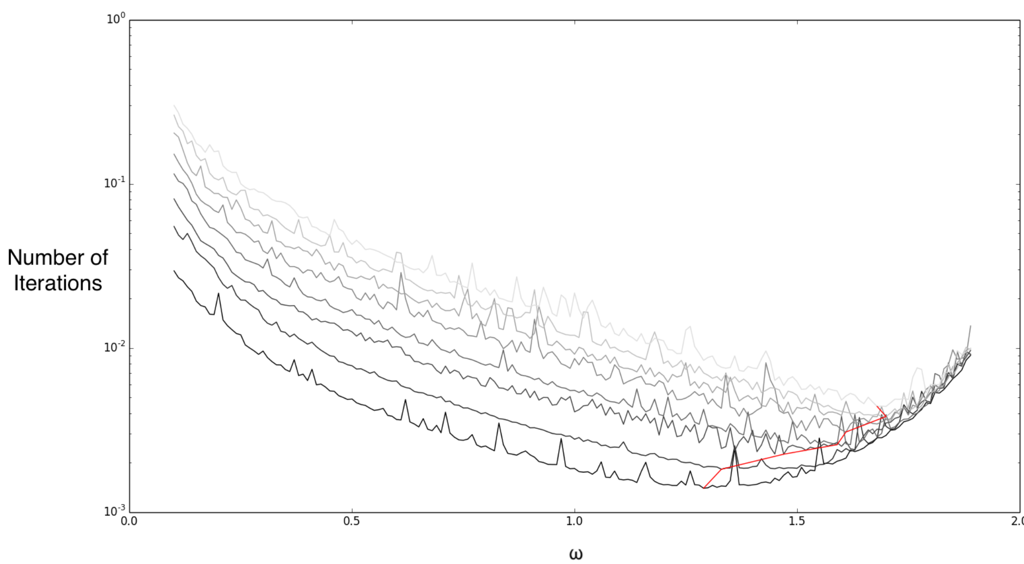

Fig. 2: Time elapsed using the matrices from Homework 5. The darkest curve represents $n = 3$, while the lightest curve represents $n = 10$. The red curve denotes the optimal ω, meaning that the algorithm runs the fastest under that ω.

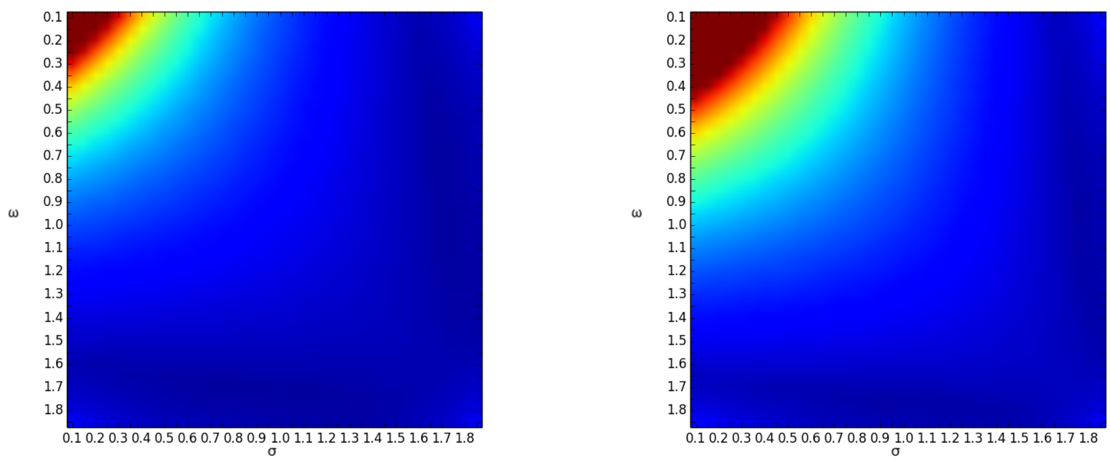

Fig. 3: Number of Iterations using the matrices from Homework 5. Left: $n = 7$. Right: $n = 9$. x-axis: σ; y-axis: ω. Dark red suggests a higher number in iteration (up to 1000), and dark blue suggests a lower number in iteration. As we can see, both plots have two "stripes" of dark blue, suggeting an optimal pair of σ, ω. As n increases, A gets larger and larger, so are the dark red region at the top left expanding further and further.

Figure 3 is a sample of two heat maps generated for $n = 7$ and $n = 9$. As n increases, the red region, which suggests a large iteration number, is expanding. The two dark blue stripes are also "squeezed" to the edge of the graph, indicating that a larger value of ω and σ is preferred when n is large.

3.3. Convergence

The USSOR method would converge if $\sigma, \omega \in (0, 2)$ and A is an SPD. More generally, if A is a non-singular H-matrix, then for all $\sigma, \omega \in \boldsymbol{O}$, USSOR would converge. Here, $\boldsymbol{O}$ is defined as: (Dragoslav & Nataa, 1993)

$$\begin{aligned}
\rho =& \rho(|L| + |U|) \\
\sigma \in& \Big(-\frac{1-\rho}{2\rho}, \frac{1+\rho}{2\rho}\Big) \\
\omega \in& \Big(max\big[\frac{|1-\sigma| + |\sigma|\rho - 1}{|1-\rho| + \rho(|\sigma|+1)}, \frac{|1-\sigma| + |\sigma|\rho - 1}{|1-\sigma|(1-\rho)}\big], \\
& min\big[\frac{1 + |1-\sigma| + |\sigma|\rho}{(1+|\sigma|)\rho + |1-\sigma|}, \frac{1 + |1-\sigma| + |\sigma|\rho}{|1-\sigma|(1+\rho)}\big]\Big)
\end{aligned}$$

4. Comparing SOR, SSOR and USSOR

The author then compare the iterations and time concumed between the three methods, using the same guideline we applied in section 2.2 and 3.2. Figure 4 is a plot of the performance of each method using optimal relaxation factor(s). The iteration curves have been normalized: since SSOR and USSOR require two sweeps of SOR, the author has multiplied the resulting iterations by 2 to reflect the real amount of iterations.

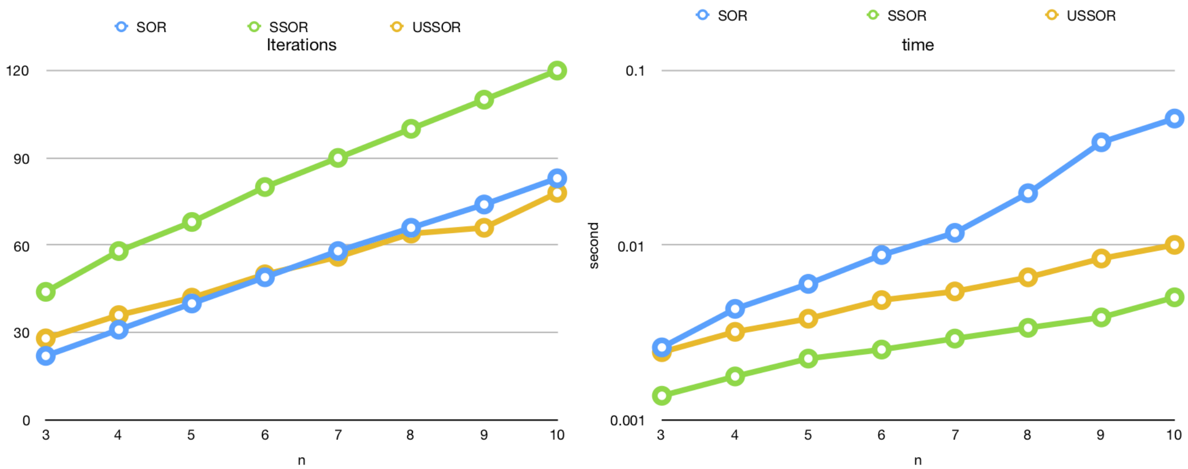

Fig. 4: Number of Iterations using the matrices from Homework 5. Left: $n = 7$. Right: $n = 9$. x-axis: σ; y-axis: ω. Dark red suggests a higher number in iteration (up to 1000), and dark blue suggests a lower number in iteration. As we can see, both plots have two "stripes" of dark blue, suggeting an optimal pair of σ, ω. As n increases, A gets larger and larger, so are the dark red region at the top left expanding further and further.

As we can see, the SOR method requires roughly same amount of iterations as USSOR, but USSOR iterates faster time-wise. SSOR has a higher iteration count, but is the fatest method amongst the three.

In an experiment solving Diritchlet problem on a unique grid, Dragoslav and Nataa obtained a similar result (Dragoslav & Nataa, 1993).

References

Sheldon, John W. (July 1955). "On the Numerical Solution of Elliptic Difference Equations". Mathematical Tables and Other Aids to Computation. 9 (51): 101. doi:10.2307/2002066

Evans, D. J.; Forrington, C. V. D. (1 November 1963). "An Iterative Process for Optimizing Symmetric Successive Over-Relaxation". The Computer Journal. 6 (3): 271273. doi:10.1093/comjnl/6.3.271

D'Sylva, E.; Miles, G. A. (1 January 1964). "The S.S.O.R. Iteration Scheme for Equations with -1 Ordering". The Computer Journal. 6 (4): 366367. doi:10.1093/comjnl/6.4.366

Young, David M. (1 January 1964). "Convergence properties of the symmetric and unsymmetric successive overrelaxation methods and related methods". Mathematics of Computation. 24 (112): 793793. doi:10.1090/S0025-5718-1970-0281331-4

Herceg, Dragoslav; Kreji, Nataa (May 1993). "On the convergence of the unsymmetric successive overrelaxation (USSOR) method". Linear Algebra and its Applications. 185: 4960. doi:10.1016/0024-3795(93)90205-3

Our class notes

```
###############################
######## SOR #################
###############################

import numpy as np
from numpy import linalg as LA
from scipy.linalg import *
import copy
import math
import time
from matplotlib import pyplot


maxIteration = 1e4
n = 0
n_dims = np.arange(3,11,1)
w_set = np.arange(0.1, 1.9, 0.01)

data = []

for k in xrange(len(n_dims)):

    n = n_dims[k]
    data += [[]]

    # generate A
    n2 = n ** 2
    A = np.diag( 4 * np.ones(n2), 0) - np.diag(np.ones(n2-1), -1) - np.diag(np.ones(n2-1),
1)\
        - np.diag(np.ones(n2-n), -n) - np.diag(np.ones(n2-n), n)

    # np.linalg.eig(A)
    print A

    # compute U, D, L
    diagA = np.diag(A)
    D = np.diag(diagA)
    U = np.triu(A) - D
    L = np.tril(A) - D

    # generate x, b
    realSolution = np.ones((n2, 1))
    b = np.dot(A, realSolution)
    x = np.zeros((n2, 1))

    for w in w_set:

        # set up parameters for GS iterations
        iterationCount = 0
        DwL = D + w * L

        old_x = np.copy(x)
        x = solve_triangular(DwL, ((1-w) * D - w * U) * old_x + w * b, lower = True)

        startTime = time.time()

        # iterate
        while (iterationCount <= maxIteration):
```

```
            old_x = np.copy(x)
            x = solve_triangular(DwL, np.dot(((1-w) * D - w * U), old_x) + w * b, lower =
True)
            iterationCount += 1
            diff = np.linalg.norm(b - np.dot(A, x), np.inf) / np.linalg.norm(b, np.inf)
            if (diff < 1e-8):
                break

        data[k] += [time.time() - startTime] #count time
        # data[k] += [iterationCount] # count iterations

    print data[k]

# record the optimal w of each dimension
optimalW = []
optimalIteration = []
for d in data:
    sdlkfj = np.argmin(d)
    optimalW += [w_set[sdlkfj]]
    optimalIteration += [d[sdlkfj]]

print
print optimalW
print optimalIteration

# plot
pyplot.yscale('log')
for j in xrange(len(data)):
    pyplot.plot(w_set, data[j], color = str(float(j)/len(data)), ls = '-')
pyplot.plot(optimalW, optimalIteration, 'r-')
pyplot.show()

###############################
######## SSOR #################
###############################

import numpy as np
from numpy import linalg as LA
from scipy.linalg import *
import copy
import math
import time
from matplotlib import pyplot

n_dims = np.arange(3,11,1)
w_set = np.arange(0.1, 1.9, 0.01)
data = []

def NORM(vector):
       return float(np.linalg.norm(vector, 2))

# tolerance and max iteration
tolerance = 1e-8
maxIteration = 1e4

for k in xrange(len(n_dims)):

       # initialization
```

```
        n = n_dims[k]
        n2 = n ** 2
        data += [[]]

        A = np.diag( 4 * np.ones(n2), 0) - np.diag(np.ones(n2-1), -1) - np.diag(np.ones(n2-1),
1)\
            - np.diag(np.ones(n2-n), -n) - np.diag(np.ones(n2-n), n)
        realSolution = np.ones((n2,1))
        b = np.dot(A, realSolution)

        # compute U, D, L
        diagA = np.diag(A)
        D = np.diag(diagA)
        U = np.triu(A) - D
        L = np.tril(A) - D

        for w in w_set:

                x = np.zeros((n2,1))

                # compute iteration matrix
                B1 = np.linalg.inv(D + w * U)
                A1 = np.dot(B1, -w * L + (1-w) * D)
                B2 = np.linalg.inv(D + w * L)
                A2 = np.dot(B2, -w * U + (1-w) * D)
                B = w * (2 - w) * np.dot(B1 , np.dot(D, np.dot(B2, b))) # w(2-w)*B1*D*B2*b
                M = np.dot(A1, A2)

                # variables
                iterationCount = 0
                solutionDifference = 1e8

                startTime = time.time()

                while (iterationCount < maxIteration and solutionDifference > tolerance):

                        iterationCount += 1

                        oldX = np.copy(x)
                        x = np.dot(M, oldX) + B

                        # print x

                        solutionDifference = NORM(b - np.dot(A, x)) / NORM(b)

                data[k] += [time.time() - startTime] #count time
                # data[k] += [iterationCount] # count iterations
        print data[k]

# record the optimal w of each dimension
optimalW = []
optimalIteration = []
for d in data:
        sdlkfj = np.argmin(d)
        optimalW += [w_set[sdlkfj]]
        optimalIteration += [d[sdlkfj]]

# print
```

```
print optimalW
print optimalIteration

# plot
pyplot.yscale('log')
for j in xrange(len(data)):
	pyplot.plot(w_set, data[j], color = str(float(j)/len(data)), ls = '-')
pyplot.plot(optimalW, optimalIteration, 'r-')
pyplot.show()

###############################
######## USSOR ################
###############################

import numpy as np
from numpy import linalg as LA
from scipy.linalg import *
import copy
import math
import time
from matplotlib import pyplot as plt

n_dims = np.arange(3,11,1)
w_set = np.arange(0.1, 1.9, 0.05)
s_set = np.arange(0.1, 1.9, 0.05)


data = []

def NORM(vector):
	return float(np.linalg.norm(vector, 2))

# tolerance and max iteration
tolerance = 1e-8
maxIteration = 1e3

for k in xrange(len(n_dims)):

	# initialization
	n = n_dims[k]
	n2 = n ** 2
	data += [[]]

	A = np.diag( 4 * np.ones(n2), 0) - np.diag(np.ones(n2-1), -1) - np.diag(np.ones(n2-1), 
1)\
	    - np.diag(np.ones(n2-n), -n) - np.diag(np.ones(n2-n), n)
	realSolution = np.ones((n2,1))
	b = np.dot(A, realSolution)

	# compute U, D, L
	diagA = np.diag(A)
	D = np.diag(diagA)
	U = np.triu(A) - D
	L = np.tril(A) - D

	# identity matrix
	E = np.identity(n2)
```

```
        for i in xrange(len(w_set)):

                w = w_set[i]
                data[k] += [[]]

                for j in xrange(len(s_set)):

                        s = w_set[j]
                        data[k][i] += [[]]

                        x = np.zeros((n2,1))

                        # compute iteration matrix
                        s_LHS = D + s * L
                        s_B = s * b
                        s_RHS = (1 - s) * D - s * U
                        w_LHS = D + w * U
                        w_B = w * b
                        w_RHS = (1 - w) * D - w * L

                        # variables
                        iterationCount = 0
                        solutionDifference = 1e8

                        startTime = time.time()

                        while (iterationCount < maxIteration and solutionDifference >
tolerance):

                                iterationCount += 1

                                oldX = np.copy(x)
                                x1 = solve_triangular(s_LHS, np.dot(s_RHS, oldX) + s_B,
lower=True)
                                x = solve_triangular(w_LHS, np.dot(w_RHS, x1) + w_B,
lower=False)

                                solutionDifference = NORM(b - np.dot(A, x)) / NORM(b)

                        # data[k] += [time.time() - startTime] #count time
                        data[k][i][j] = iterationCount # count iterations


        fig, ax = plt.subplots()
        im = ax.imshow(data[k], vmin=0, vmax=maxIteration)

        # We want to show all ticks...
        ax.set_xticks(np.arange(len(s_set)))
        ax.set_yticks(np.arange(len(w_set)))
        # ... and label them with the respective list entries
        ax.set_xticklabels(s_set)
        ax.set_yticklabels(w_set)

        # make some ticks invisible or it will become unreadable otherwise
        for label in ax.xaxis.get_ticklabels()[1::2]:
            label.set_visible(False)

        for label in ax.yaxis.get_ticklabels()[1::2]:
```

```
            label.set_visible(False)


        fig.tight_layout()
        plt.savefig('USSOR_iterations_' + str(n) + '.png')
        # plt.show()

print data
```