Problem 1

This MATLAB script computes the pseudoinverse of a matrix A using SVD. First the code computes the SVD of A. It then takes the diagonal elements of S and over the next few lines creates the inverse of S. From here it multiplies $VS^{-1}U^T$ to compute the pseudoinverse of A.

Problem 2

Consider the following matrices

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & -2 \end{bmatrix}, \; B = \begin{bmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix}$$

To test if these are convergent for a given method, we much show that $p(G) < 1$, where $G = MN^{-1}$. For the Jacobi method, $G = D^{-1}(E + F)$, so we find that

$$G_A = \begin{bmatrix} 0 & 1/2 & -1/2 \\ -1 & 0 & -1 \\ 1/2 & 1/2 & 0 \end{bmatrix}, \; G_B = \begin{bmatrix} 0 & -2 & 2 \\ -1 & 0 & -1 \\ -2 & -2 & 0 \end{bmatrix}$$

Thus, we find that $p(G_A) = \frac{\sqrt{5}}{2}$ and $p(G_B) = 0$, so by the theorem from class, the matrix A does not converge and matrix B does.

For Gauss-Seidel, $G = (D - E)^{-1}F$, so we find that

$$G_A = \begin{bmatrix} 0 & 1/2 & -1/2 \\ 0 & -1/2 & -1/2 \\ 0 & 0 & -1/2 \end{bmatrix} G_B = \begin{bmatrix} 0 & -2 & 2 \\ 0 & 2 & -3 \\ 0 & 0 & 2 \end{bmatrix}$$

Thus, we find that $p(G_A) = \frac{1}{2}$ and $p(G_B) = 2$, so by the theorem matrix A will be convergent and matrix B will not be.

Problem 3

Consider the residual of the steepest decent method, so $r_j = b - Ax^{(j)}$. Thus

$$\begin{aligned} r_{j+1} &= b - Ax^{(j+1)} \\ &= b - A(x^{(j)} + \alpha r) \\ &= b - A(x^{(j)} + \frac{r^T r}{r^T Ar} r) \\ &= b - Ax^{(j)} - A\frac{r^T r}{r^T Ar} r \end{aligned}$$

We wish to show that $\langle r_j, r_{j+1} \rangle = 0$. Thus,

$$\begin{aligned}
\langle r_j, r_{j+1} \rangle &= (b - Ax^{(j)})^T (b - Ax^{(j)} - A\frac{r_j^T r_j}{r_j^T A r_j} r_j) \\
&= r_j^T r_j - (b - Ax^{(j)})^T A \frac{r_j^T r_j}{r_j^T A r_j} r_j \\
&= r_j^T r_j - r_j^T A \frac{r_j^T r_j}{r_j^T A r_j} r_j \\
&= r_j^T r_j - r_j^T r_j \left(\frac{r_j^T A r_j}{r_j^T A r_j} \right) \\
&= r_j^T r_j - r_j^T r_j * 1 \\
&= 0
\end{aligned}$$

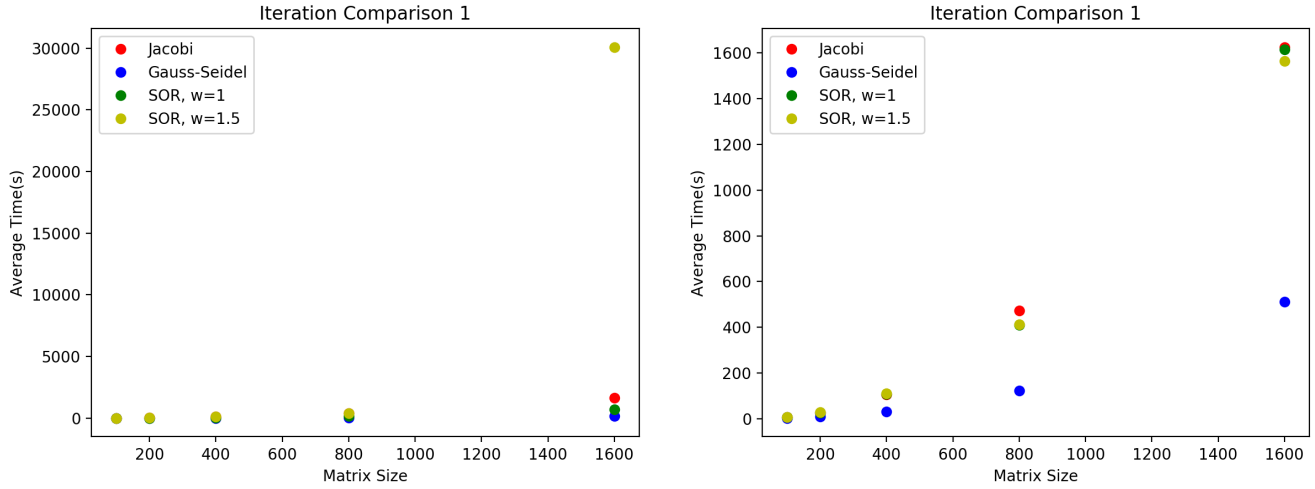
□

Problem 4

Please note that all code being references for problems 4 and 5 may be found at the end of this document.

This problem involved solving $T + \mu I$ matrices of various sizes using the Jacobi, Gauss-Seidel, and SOR methods. For each computation, the maximum number of iterations allowed were 50 and the desired error of the solutions were 10^{-10}. For the SOR methods, the relaxation parameter was either $w = 1$ or $w = 1.5$. Results for this can be found below:

Figure 1: Comparing various Iterative Methods for $\mu = 5$ and $\mu = -5$ respectively



While computing these solutions, the most striking difference was the speed. For $\mu = 5$, the computations were speedy, especially for the Gauss-Seidel and $\text{SOR}_{\text{w = 1.5}}$ methods, which would terminate before all iterations were necessary. Contrasting this, for $\mu = -5$ none of the computations terminated early, which likely means they did not converge to the error desired. According to these results, for $\mu = 5$, the fastest method was GS with $\text{SOR}_{\text{w = 1}}$ a close second. The slowest method was $\text{SOR}_{\text{w = 1.5}}$. These results make sense, as GS is known to be fast. Additionally,

since I did not optimize the relaxation parameter and instead chose them at random, the drastic speed difference between the two SOR instances makes sense. Had an optimized relaxation parameter been utilized, SOR would no doubt have been the fastest. For $\mu = -5$, the solutions took far longer to compute. I imagine this it due to many of the vectors not converging, which may be a result of the μ value changing the spectral radius of the matrices. Additionally, this change in matrix led to the relaxation parameter 1.5 to be more optimized for the problem, while the relaxation parameter 1.0 was less optimized. Regardless, the GS method still reigned supreme in speed compared to other results.

Problem 5

For this problem, I was tasked to solve the following differential equation using the Steepest Descent and Conjugate Gradient methods:

$$\begin{cases} -u'' + \lambda u = 3x - \frac{1}{2} & x \in [0,1] \\ u(0) = 0, \ u(1) = -2 \end{cases}$$

This task proved to be more difficult than expected. For whatever reason, the solutions vectors for this problem would not converge, despite great refinement of the mesh. In this, 2000 iterations were done with an error cutoff of 10^{-5} (which was never obtained in the 2000 iterations). Because of this, the algorithm solutions do not match the "true" solution exactly, as seen in the figures below. I found my code to be inconsistent; sometimes it would accurately match the true solution and sometimes it would be completely off. I believe this greatly depends on the size of the matrix and the number of iterations, as smaller matrices actually led to more accurate results.

Figure 2: Time Comparison between methods

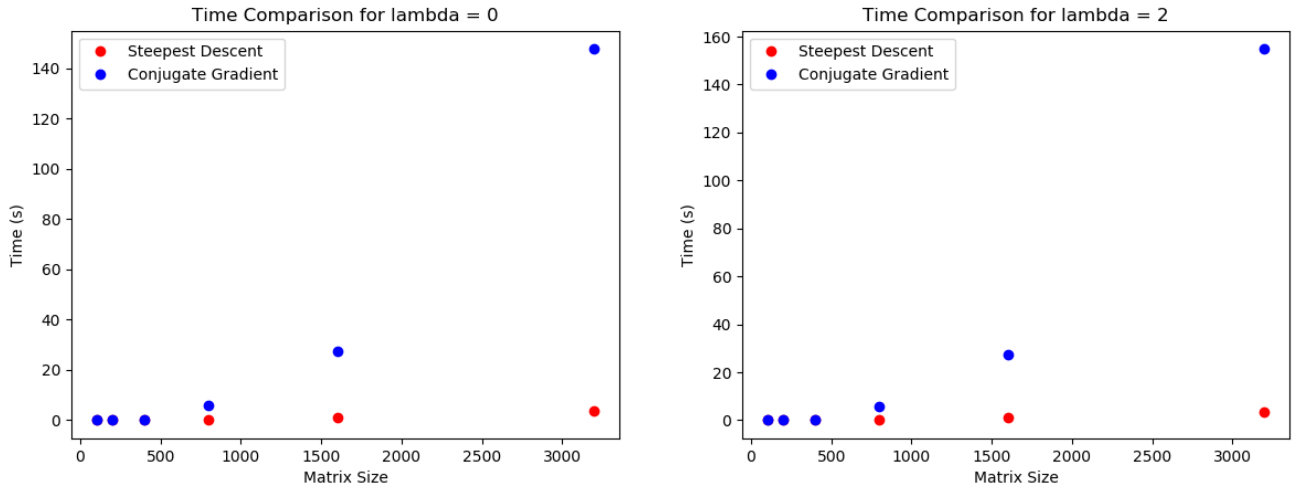


Figure 3: Comparing Steepest Descent to Exact Solution

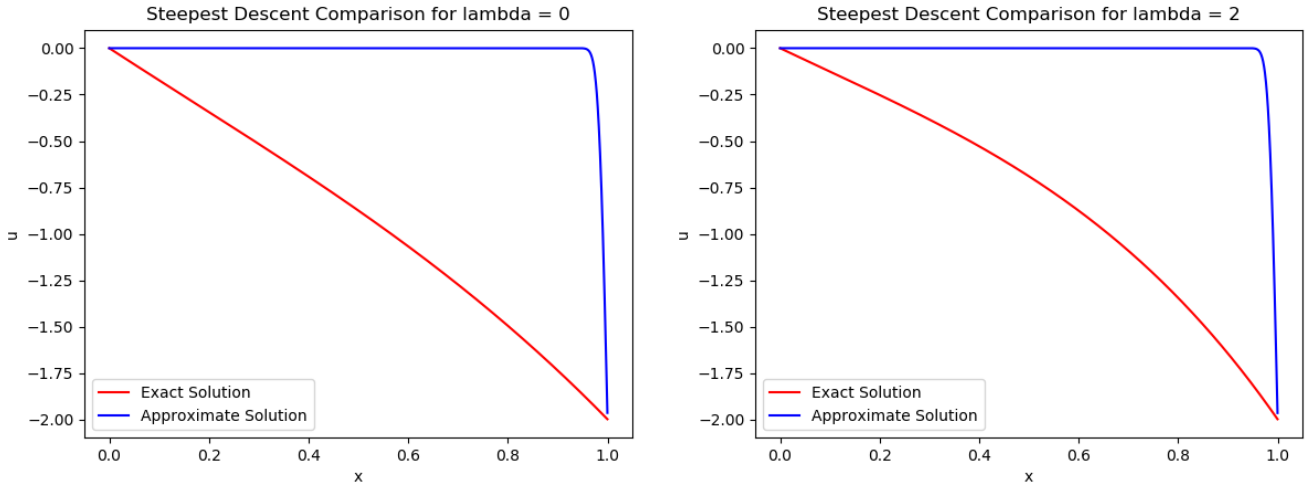


Figure 4: Comparing Conjugate Gradient to Exact Solution

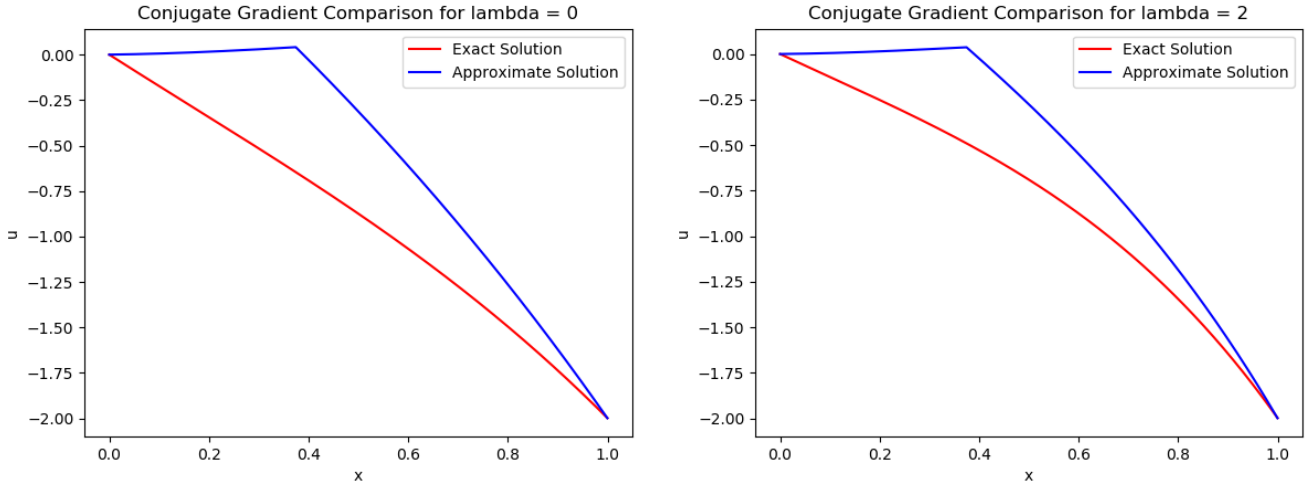
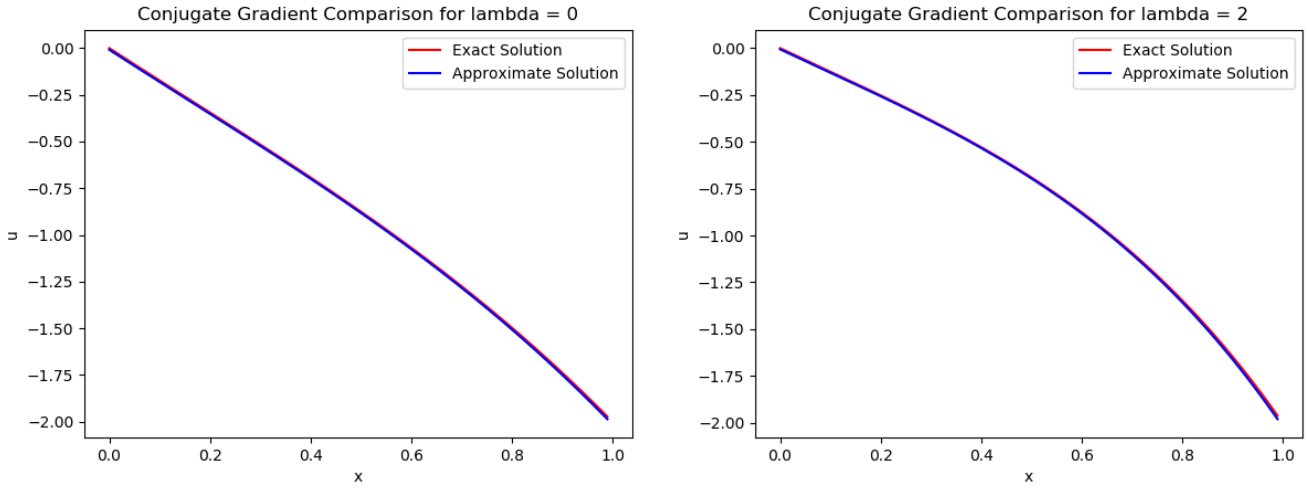


Figure 5: Conjugate Gradient Method for 200 x 200 matrix with 2000 iterations



As seen above, for certain parameters the conjugate gradient method was able to approximate the exact solution well, but for other parameters not so much. On the other hand, steepest descent never converged and thus did not achieve results near the exact solution. However, steepest descent did run far faster than conjugate gradient as seen in Figure 1.

Problem Code

```
import numpy as np


"""
This file holds iteration methods for solving Ax = b
"""


"""
The Jacobi method solves Ax = b
Inputs:
    A    - The matrix
    x    - The initial guess vector
    b    - The vector
    err  - The error desired for the solution
    T    - The number of iterations to be done
"""
def jacobi(A,x,b,err,T):
    n = A.shape[0]
    count = 0
    while (count < T+1):
```

```
        if count % 10 == 0:
            print count
        u = np.matrix(np.zeros(n)).T
        for i in range(n):
            s = 0.0
            for j in range(n):
                if j != i:
                    s += float(A[i,j]*x[j])
            u[i] = b[i] - s
            u[i] /= float(A[i,i])

        x = u
        if (np.linalg.norm(b - A*x) < err):
            return x,count
        count += 1

    return x,count

def gs(A,x,b,err,T):
    n = A.shape[0]
    count = 0
    x = x.astype(float)
    while (count < T+1):
        if count % 10 == 0:
            print count
        for i in range(n):
            s = 0.0
            for j in range(n):
                if j != i:
                    s += float(A[i,j])*float(x[j,0])
            k = (b[i,0] - s) / float(A[i,i])
            x[i,0] = k

        if (np.linalg.norm(b - A*x) < err):
            return x,count
        count += 1

    return x,count
```

```
def sor(A,x,b,w,err,T):
    n = A.shape[0]
    count = 0
    x = x.astype(float)
    while (count < T+1):
        if count % 10 == 0:
            print count
        for i in range(n):
            s = 0.0
            for j in range(n):
                if j != i:
                    s += float(A[i,j]*x[j])
            x[i] = (1-w)*x[i] + w*float(b[i] - s)/float(A[i,i])

        if (np.linalg.norm(b - A*x) < err):
            return x,count
        count += 1

    return x,count

def steepestdescent(A,x,b,err,T):
    count = 0
    x = x.astype(float)
    while (count < T+1):
        if count % 10 == 0:
            print count
        r = b - A*x
        alpha = float(r.T*r / (r.T*A*r))
        x = x + alpha * r
        if (np.linalg.norm(b - A*x) < err):
            return x, count
        count += 1
    return x, count

def gradiantdescent(A,x,b,err,T):
    count = 0
    x = x.astype(float)
```

```
    r = b - A*x
    p = r
    while (count < T+1):
        if count % 10 == 0:
            print count
        alpha = float(r.T*r / (p.T*A*p))
        x = x + alpha * p
        t = 1/float(r.T*r)
        r = r - alpha * A * p
        p = r + float(r.T*r) * t * p
        if (np.linalg.norm(b - A*x) < err):
            return x, count
        count += 1
    return x, count

import numpy as np

"""
A program that stores matrix generation functions
for various homework problems
"""

#computes a random tridiagonal matrix
def HW4P4(m):
    A = np.matrix([[0.0 for x in range(int(m))] for y in range(int(m))])
    for i in range (m):
        A[i,i] = 10
        if i < m-1:
            A[i+1,i] = 3
            A[i,i+1] = 3
        if i < m-2:
            A[i+2,i] = 2
            A[i,i+2] = 2
        if i < m-3:
            A[i+3,i] = 1
            A[i,i+3] = 1
    return A
```

```
#computes a central difference matrix
def HW4P5(n,val,alpha,beta, ualpha, ubeta):
    A = np.matrix(np.identity(n))
    b = np.matrix(np.zeros(n)).T
    h = (beta - alpha) / float((n))
    for i in range(0,n):
        A[i,i] = 2.0 + val * h * h
        if (i + 1 < n):
            A[i+1,i] = -1.0
            A[i,i+1] = -1.0
        if (i == 0):
            b[i,0] = h*h*func(alpha + (i+1) * h) + ualpha
        elif (i == n-1):
            b[i,0] = h*h*func(alpha + (i+1) * h) + ubeta
        else:
            b[i,0] = h*h*func(alpha + (i+1) * h)

    return A, b

def func(x):
    return 3.0*x - 0.5
```