

# Numerical Analysis Computational Assignment B

**Name:** Nate Stemen (20906566)  
**Email:** nate.stemen@uwaterloo.ca

**Due:** Fri, Sep 25, 2020 5:00 PM  
**Course:** AMATH 740

## Problem 1. Implementation of the Gauss-Seidel method.

**Solution.** First, the code from GaussSeidel.py.

```
1 import numpy as np
2
3
4 def GaussSeidel(A, guess, b, tolerance, maxIterations):
5     n = A.shape[0]
6     x_old = guess
7     for iteration in range(maxIterations):
8         x_new = np.zeros_like(x_old)
9         for i in range(n):
10             new_sum = np.dot(A[i, :], x_new[:i])
11             old_sum = np.dot(A[i, i + 1 :], x_old[i + 1 :])
12             x_new[i] = (b[i] - new_sum - old_sum) / A[i, i]
13             if np.linalg.norm(x_new - x_old) < tolerance:
14                 break
15         x_old = x_new
16     if iteration == maxIterations - 1:
17         print("HIT MAX ITERATION: ", maxIterations)
18     return x_new
```

And here's the output of VerifyGaussSeidel.py:

total error: 7.596379365363873e-14

So pretty good if I do say so myself. And because I'm working in python here's the file VerifyGaussSeidel.py.

```
1 import numpy as np
2 from GaussSeidel import GaussSeidel
3
4 n = 100
5
6 A = np.random.rand(n, n)
7 b = np.random.rand(n)
8
9 for i in range(n):
10     A[i, i] = sum(A[i, j] for j in range(n))
11
12 x = GaussSeidel(A, np.zeros(n), b, 1e-12, 1000)
13
14 error = np.linalg.norm(x - np.linalg.solve(A, b))
15
16 print(f"total error: {error}")
```

**Problem 4.** Implementation of the preconditioned CG and GMRES methods.

**Solution.** First, the code from myGMRES.py:

```

1 import numpy as np
2 from scipy import sparse, linalg
3
4
5 def myGMRES(A, guess, b, tolerance=1e-12, maxIterations=1000):
6     n = A.shape[0]
7     maxIterations = min(n, maxIterations)
8     r0 = b - A @ guess
9     rho = np.linalg.norm(r0)
10    Q = np.zeros((n, maxIterations))
11    Q[:, 0] = r0 / rho
12    H = sparse.lil_matrix((maxIterations + 1, maxIterations))
13    residuals = []
14    for iteration in range(maxIterations):
15        v = A @ Q[:, iteration]
16        for j in range(iteration + 1):
17            H[j, iteration] = np.dot(Q[:, j], v)
18            v -= H[j, iteration] * Q[:, j]
19        norm_v = np.linalg.norm(v)
20        H[iteration + 1, iteration] = norm_v
21        Q[:, iteration + 1] = v / norm_v
22
23        e = np.zeros(maxIterations + 1)
24        e[0] = rho
25
26        y, *_ = linalg.lstsq(H.toarray(), e)
27        residual = np.linalg.norm(e - H @ y) / rho
28
29        residuals.append(residual)
30        if residual < tolerance or iteration == maxIterations - 1:
31            x = guess + Q @ y
32            break
33
34    return x, residuals, iteration + 1

```

Second, the code from myGMRES\_SSOR.py:

```

1 import numpy as np
2 from scipy import sparse, linalg
3
4
5 def myGMRES_SSOR(A, guess, b, tolerance=1e-12, maxIterations=1000):
6     n = A.shape[0]
7     maxIterations = min(n, maxIterations)
8     omega = 1.9
9     AL = -1 * sparse.tril(A, k=-1)
10    AU = -1 * sparse.triu(A, k=1)
11    AD = sparse.spdiags(A.diagonal(), [0], n, n)
12    ADinv = sparse.spdiags([1 / x for x in A.diagonal()], [0], n, n)
13    Pinv = (AD - omega * AL) @ ADinv @ (AD - omega * AU) / (omega * (2
        - omega))
14
15    r0 = b - A @ guess
16    rho = np.linalg.norm(r0)
17    r0 /= rho
18
19    Q = np.zeros((n, maxIterations))
20    Q[:, 0] = r0
21    H = sparse.lil_matrix((maxIterations + 1, maxIterations))
22    residuals = []
23    for iteration in range(maxIterations):
24        z = sparse.linalg.spsolve(Pinv, Q[:, iteration])
25        v = A @ z
26        for j in range(iteration + 1):
27            H[j, iteration] = np.dot(Q[:, j], v)
28            v -= H[j, iteration] * Q[:, j]
29
30        norm_v = np.linalg.norm(v)
31        H[iteration + 1, iteration] = norm_v
32        Q[:, iteration + 1] = v / norm_v
33
34        e = np.zeros(maxIterations + 1)
35        e[0] = rho
36        y, *_ = linalg.lstsq(H.toarray(), e)
37        residual = np.linalg.norm(e - H @ y) / rho
38        residuals.append(residual)
39
40        if residual < tolerance or iteration == n - 1:
41            w = sparse.linalg.spsolve(Pinv, Q @ y)
42            x = guess + w
43            break
44
45    return x, residuals, iteration + 1

```

Third, the code from myCG\_SSOR.py:

```

1 import numpy as np
2 from scipy import sparse
3 from scipy.sparse import linalg
4
5
6 def myCG_SSOR(A, guess, b, tolerance=1e-12, maxIterations=1000):
7     n = A.shape[0]
8     maxIterations = min(n, maxIterations)
9     omega = 1.9
10    AL = -1 * sparse.tril(A, k=-1)
11    AU = -1 * sparse.triu(A, k=1)
12    AD = sparse.spdiags(A.diagonal(), [0], n, n)
13    ADinv = sparse.spdiags([1 / x for x in A.diagonal()], [0], n, n)
14    Pinv = (AD - omega * AL) @ ADinv @ (AD - omega * AU) / (omega * (2
        - omega))
15
16    Rvecs = np.zeros((n, maxIterations))
17    Qvecs = np.zeros((n, maxIterations))
18    Pvecs = np.zeros((n, maxIterations))
19    Xvecs = np.zeros((n, maxIterations))
20    Xvecs[:, 0] = guess
21
22    r0 = b - A @ guess
23    q0 = linalg.spsolve(Pinv, r0)
24    p0 = np.copy(q0)
25    Rvecs[:, 0], Qvecs[:, 0], Pvecs[:, 0] = r0, q0, p0
26
27    residuals = [np.linalg.norm(r0)]
28    for k in range(1, maxIterations):
29        Aonp = A @ Pvecs[:, k - 1]
30        alpha = np.dot(Rvecs[:, k - 1], Qvecs[:, k - 1]) / np.dot(Pvecs
           [:, k - 1], Aonp)
31
32        xk = Xvecs[:, k - 1] + alpha * Pvecs[:, k - 1]
33        rk = Rvecs[:, k - 1] - alpha * Aonp
34
35        residual = np.linalg.norm(rk) / residuals[0]
36        residuals.append(residual)
37        if residual < tolerance or k == maxIterations - 1:
38            return xk, residuals, k
39
40        Xvecs[:, k] = xk
41        Rvecs[:, k] = rk
42
43        qk = sparse.linalg.spsolve(Pinv, rk)
44        Qvecs[:, k] = qk
45
46        beta = np.dot(rk, qk) / np.dot(Rvecs[:, k - 1], Qvecs[:, k -
            1])
47        Pvecs[:, k] = qk + beta * Pvecs[:, k - 1]
48    return xk, residuals, maxIterations

```

I also wrote a version of Conjugate Gradient without preconditioning, so here's that:

```

1 import numpy as np
2 from scipy import sparse
3
4
5 def myCG(A, guess, b, tolerance=1e-12, maxIterations=1000):
6     n = A.shape[0]
7     maxIterations = min(n, maxIterations)
8
9     Rvecs = np.zeros((n, maxIterations))
10    Pvecs = np.zeros((n, maxIterations))
11    Xvecs = np.zeros((n, maxIterations))
12    Xvecs[:, 0] = guess
13
14    r0 = b - A @ guess
15    p0 = np.copy(r0)
16    Rvecs[:, 0], Pvecs[:, 0] = r0, p0
17
18    residuals = [np.linalg.norm(r0)]
19    for k in range(1, maxIterations):
20        Aonp = A @ Pvecs[:, k - 1]
21        alpha = np.dot(Rvecs[:, k - 1], Rvecs[:, k - 1]) / np.dot(Pvecs
22           [:, k - 1], Aonp)
23
24        xk = Xvecs[:, k - 1] + alpha * Pvecs[:, k - 1]
25        rk = Rvecs[:, k - 1] - alpha * Aonp
26
27        residual = np.linalg.norm(rk) / residuals[0]
28        residuals.append(residual)
29        if residual < tolerance or k == maxIterations - 1:
30            return xk, residuals, k
31
32        Xvecs[:, k] = xk
33        Rvecs[:, k] = rk
34
35        beta = np.dot(rk, rk) / np.dot(Rvecs[:, k - 1], Rvecs[:, k -
36            1])
37        Pvecs[:, k] = rk + beta * Pvecs[:, k - 1]
38    return xk, residuals, maxIterations

```

Now, here's the output of test\_iterative.py:

```

*****
error for GMRES, # of steps
error: 4.164109009000164e-13
steps: 31
----
error for GMRES_SSOR, # of steps
error: 2.6580413851364233e-12
steps: 22
----
error for CG, # of steps
error: 4.1577704537099615e-13
steps: 31
----
error for CG_SSOR, # of steps

```

error: 1.904849842831412e-12

steps: 23

----

And the translated code test\_iterative.py:

```

1 import numpy as np
2 import sys
3
4 sys.path.insert(0, "../assignment1/problemThree")
5 from build_laplace_2d import build_laplace_2D
6 from myGMRES import myGMRES
7 from myGMRESSSOR import myGMRES_SSOR
8 from myCG import myCG
9 from myCGSSOR import myCG_SSOR
10
11 print("*****")
12
13 maxIterations = 500
14 tolerance = 1e-12
15 N = 8
16 n = N ** 2
17 A = build_laplace_2D(N)
18 x_exact = np.random.rand(n)
19 b = A @ x_exact
20 x0 = np.zeros(n)
21
22 x_gmres, resgmres, stepsgmres = myGMRES(A, x0, b, tolerance,
    maxIterations)
23 x_gmresSSOR, resgmresSSOR, stepsgmresSSOR = myGMRES_SSOR(
24     A, x0, b, tolerance, maxIterations
25 )
26 x_cg, rescg, stepscg = myCG(A, x0, b, tolerance, maxIterations)
27 x_cgSSOR, rescgSSOR, stepscgSSOR = myCG_SSOR(A, x0, b, tolerance,
    maxIterations)
28
29 print("error for GMRES, # of steps")
30 error = np.linalg.norm(x_exact - x_gmres)
31 print(f"error: {error}")
32 print(f"steps: {stepsgmres}")
33
34 print("----")
35
36 print("error for GMRES_SSOR, # of steps")
37 error = np.linalg.norm(x_exact - x_gmresSSOR)
38 print(f"error: {error}")
39 print(f"steps: {stepsgmresSSOR}")
40
41 print("----")
42
43 print("error for CG, # of steps")
44 error = np.linalg.norm(x_exact - x_cg)
45 print(f"error: {error}")
46 print(f"steps: {stepscg}")
47
48 print("----")
49
50 print("error for CG_SSOR, # of steps")

```

```

51 error = np.linalg.norm(x_exact - x_cgSSOR)
52 print(f"error: {error}")
53 print(f"steps: {stepscgSSOR}")

```

(e)

Now here's the code for driverPCG\_PGMRES.py:

```

1  import numpy as np
2  import sys
3
4  sys.path.insert(0, "../assignment1/problemThree")
5
6  from build_laplace_2d import build_laplace_2D
7  from myGMRES import myGMRES
8  from myGMRESSSOR import myGMRES_SSOR
9  from myCG import myCG
10 from myCGSSOR import myCG_SSOR
11
12 import matplotlib.pyplot as plt
13
14 plt.style.use("ggplot")
15
16 maxIterations = 400
17 tolerance = 1e-10
18
19 N = 32
20 n = N ** 2
21 A = build_laplace_2D(N)
22
23 b = np.ones(n)
24
25 x0 = np.zeros(n)
26
27 x_gmres, resgmres, stepsgmres = myGMRES(A, x0, b, tolerance,
28                                         maxIterations)
29 x_gmresSSOR, resgmresSSOR, stepsgmresSSOR = myGMRES_SSOR(
30     A, x0, b, tolerance, maxIterations
31 )
32 x_cg, rescg, stepscg = myCG(A, x0, b, tolerance, maxIterations)
33 x_cgSSOR, rescgSSOR, stepscgSSOR = myCG_SSOR(A, x0, b, tolerance,
34                                                maxIterations)
35
36 print(f"Steps for GMRES:           {stepsgmres}")
37 print("----")
38 print(f"Steps for GMRES with SSOR: {stepsgmresSSOR}")
39 print("----")
40 print(f"Steps for CG:                 {stepscg}")
41 print("----")
42 print(f"Steps for CG with SSOR:      {stepscgSSOR}")
43
44 fig = plt.figure()
45 ax = plt.subplot(111)
46 ax.plot(resgmres, label="GMRES")
47 ax.plot(resgmresSSOR, label="GMRES with SSOR")
48 ax.plot(rescg, label="CG")
49 ax.plot(rescgSSOR, label="CG with SSOR")

```

```

48
49 plt.title("Residual Size")
50 ax.legend()
51 ax.set_xlabel(r"Iterations $n$")
52 ax.set_ylabel(r"Residual $\|r_n\|$")
53 plt.yscale("log")
54 plt.show()

```

And it's output:

Steps for GMRES: 66

----

Steps for GMRES with SSOR: 31

----

Steps for CG: 66

----

Steps for CG with SSOR: 32

And the associated plot. Really cool to see the two preconditioned algorithms perform-

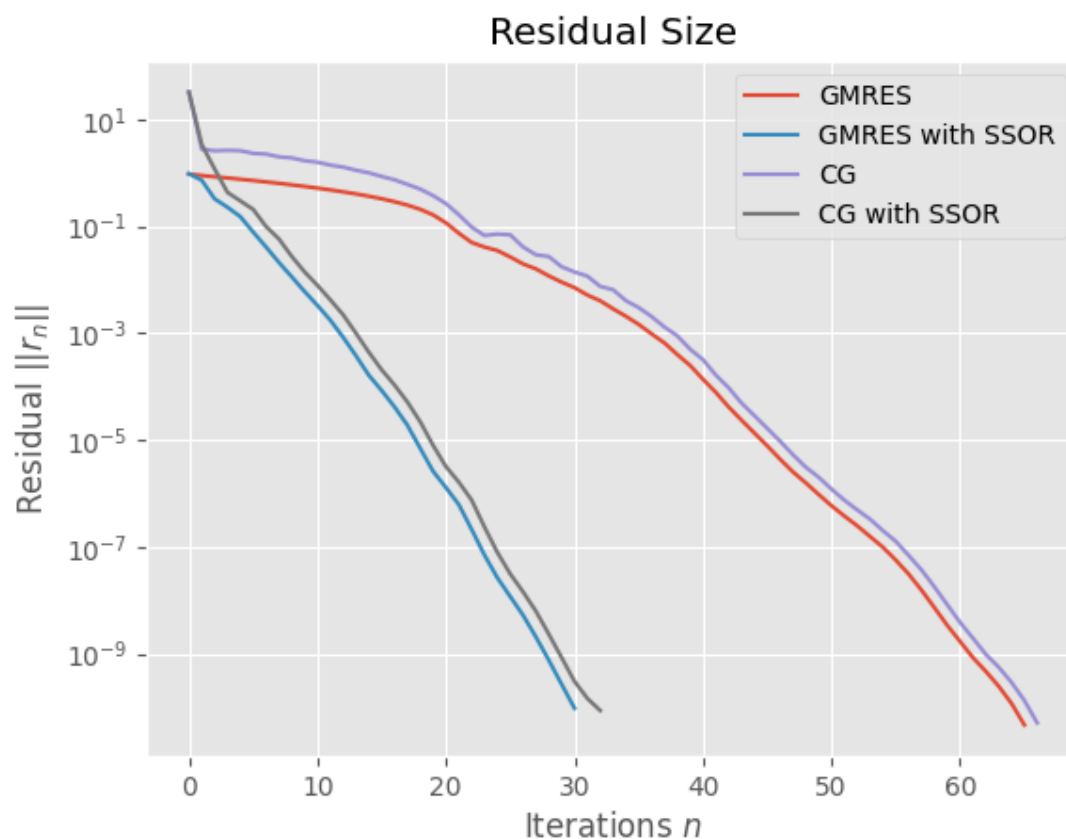


Figure 4.1: Residual Size as a function of iteration number

ing so much better on this problem. I would never have expected, but then again, what do I know!? It says in the homework CG with SSOR is the optimal solution for this problem, but GMRES with SSOR seems to be performing slightly better. Why is that?



(f)

For this part I wrote a new file to generate the required log-log plot. Here's `loglog.py`:

```

1 import numpy as np
2 import sys
3
4 sys.path.insert(0, "../assignment1/problemThree")
5 from build_laplace_2d import build_laplace_2D
6 from myGMRES import myGMRES
7 from myGMRESSSOR import myGMRES_SSOR
8 from myCG import myCG
9 from myCGSSOR import myCG_SSOR
10 import matplotlib.pyplot as plt
11
12 plt.style.use("ggplot")
13
14 Ns = range(80, 150, 10)
15 ns = [N ** 2 for N in Ns]
16 tol = 1e-10
17
18 steps = {"gmres": [], "gmresssor": [], "cg": [], "cgssor": []}
19 for N in Ns:
20     print(N)
21     n = N ** 2
22     A = build_laplace_2D(N)
23     b = np.ones(n)
24     x0 = np.zeros(n)
25
26     _, stepsgmres = myGMRES(A, x0, b, tol)
27     print("\tdone with GMRES")
28     _, stepsgmresSSOR = myGMRES_SSOR(A, x0, b, tol)
29     print("\tdone with GMRES SSOR")
30     _, stepscg = myCG(A, x0, b, tol)
31     print("\tdone with CG")
32     _, stepscgSSOR = myCG_SSOR(A, x0, b, tol)
33     print("\tdone with CG SSOR")
34
35     steps["gmres"].append(stepsgmres)
36     steps["gmresssor"].append(stepsgmresSSOR)
37     steps["cg"].append(stepscg)
38     steps["cgssor"].append(stepscgSSOR)
39
40 fig = plt.figure()
41 ax = plt.subplot(111)
42 x = np.log(ns)
43 ax.plot(x, np.log(steps["gmres"]), label="GMRES")
44 ax.plot(x, np.log(steps["gmresssor"]), label="GMRES with SSOR")
45 ax.plot(x, np.log(steps["cg"]), label="CG")
46 ax.plot(x, np.log(steps["cgssor"]), label="CG with SSOR")
47
48 ax.legend()
49 ax.set_xlabel(r"Log of Problem Size")
50 ax.set_ylabel(r"Log of # of Iterations")
51 plt.show()
52
53 slope_gmres, _ = np.polyfit(x, np.log(steps["gmres"]), 1)
54 print(f"GMRES loglog slope: {slope_gmres}")
55 slope_cg, _ = np.polyfit(x, np.log(steps["cg"]), 1)

```

```

56 print(f"CG loglog slope: {slope_cg}")
57 slope_gmresssor, _ = np.polyfit(x, np.log(steps["gmresssor"]), 1)
58 print(f"GMRES with SSOR loglog slope: {slope_gmresssor}")
59 slope_cgssor, _ = np.polyfit(x, np.log(steps["cgssor"]), 1)
60 print(f"CG with SSOR loglog slope: {slope_cgssor}")

```

This generates the following log-log plot. Note both CG and GMRES (without preconditioning) lie basically on top of each other, and are hard to distinguish.

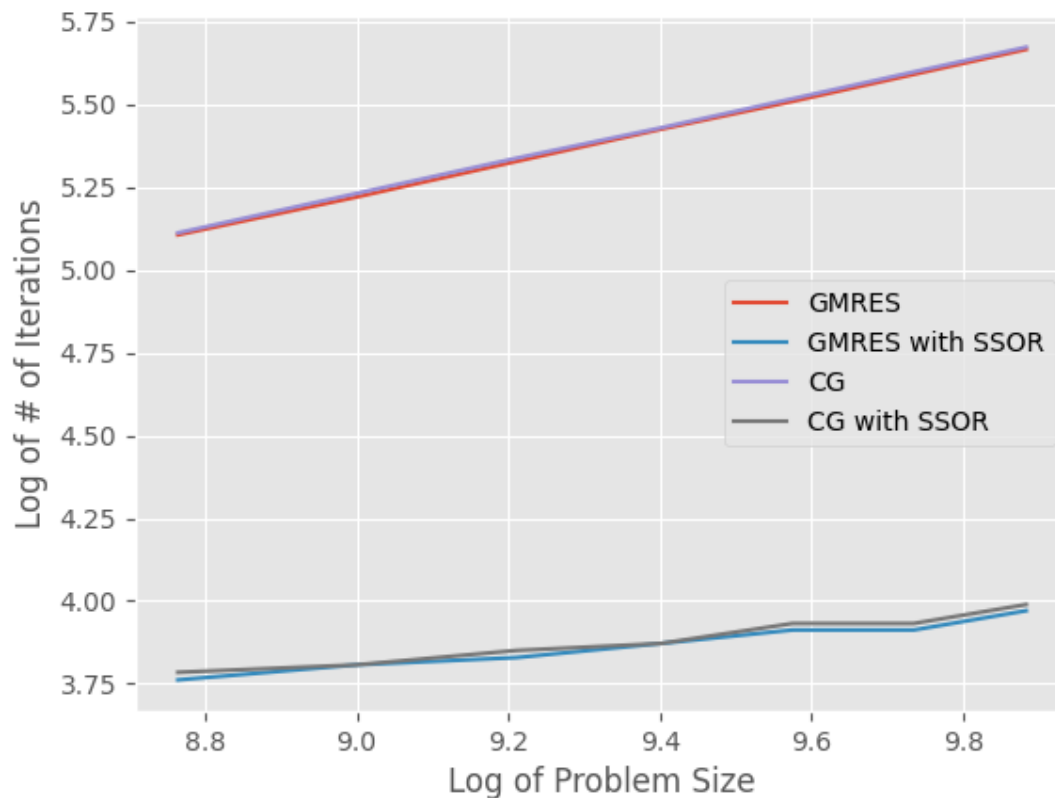


Figure 4.2: Log-log plot

The file also prints out the slopes of these lines as given by `np.polyfit`.

```

GMRES loglog slope: 0.5013427593113042
CG loglog slope: 0.5003337819405206
GMRES with SSOR loglog slope: 0.17676643756871577
CG with SSOR loglog slope: 0.18072452126753552

```

With this we see that both CG and GMRES have a slope of approximately  $\frac{1}{2}$  and the preconditioned problems have slope that's about three times as small. Pretty sweet!