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`.

```
import numpy as np


def GaussSeidel(A, guess, b, tolerance, maxIterations):
    n = A.shape[0]
    x_old = guess
    for iteration in range(maxIterations):
        x_new = np.zeros_like(x_old)
        for i in range(n):
            new_sum = np.dot(A[i, :i], x_new[:i])
            old_sum = np.dot(A[i, i + 1 :], x_old[i + 1 :])
            x_new[i] = (b[i] - new_sum - old_sum) / A[i, i]
        if np.linalg.norm(x_new - x_old) < tolerance:
            break
        x_old = x_new
    if iteration == maxIterations - 1:
        print("HIT MAX ITERATION: ", maxIterations)
    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`.

```
import numpy as np
from GaussSeidel import GaussSeidel

n = 100

A = np.random.rand(n, n)
b = np.random.rand(n)

for i in range(n):
    A[i, i] = sum(A[i, j] for j in range(n))

x = GaussSeidel(A, np.zeros(n), b, 1e-12, 1000)

error = np.linalg.norm(x - np.linalg.solve(A, b))

print(f"total error: {error}")
```

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

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

```python
import numpy as np
from scipy import sparse, linalg


def myGMRES(A, guess, b, tolerance=1e-12, maxIterations=1000):
    n = A.shape[0]
    maxIterations = min(n, maxIterations)
    r0 = b - A @ guess
    rho = np.linalg.norm(r0)
    Q = np.zeros((n, maxIterations))
    Q[:, 0] = r0 / rho
    H = sparse.lil_matrix((maxIterations + 1, maxIterations))
    residuals = []
    for iteration in range(maxIterations):
        v = A @ Q[:, iteration]
        for j in range(iteration + 1):
            H[j, iteration] = np.dot(Q[:, j], v)
            v -= H[j, iteration] * Q[:, j]
        norm_v = np.linalg.norm(v)
        H[iteration + 1, iteration] = norm_v
        Q[:, iteration + 1] = v / norm_v

        e = np.zeros(maxIterations + 1)
        e[0] = rho

        y, *_ = linalg.lstsq(H.toarray(), e)
        residual = np.linalg.norm(e - H @ y) / rho

        residuals.append(residual)
        if residual < tolerance or iteration == maxIterations - 1:
            x = guess + Q @ y
            break

    return x, residuals, iteration + 1
```

Second, the code from `myGMRES_SSOR.py`:

```
import numpy as np
from scipy import sparse, linalg


def myGMRES_SSOR(A, guess, b, tolerance=1e-12, maxIterations=1000):
    n = A.shape[0]
    maxIterations = min(n, maxIterations)
    omega = 1.9
    AL = -1 * sparse.tril(A, k=-1)
    AU = -1 * sparse.triu(A, k=1)
    AD = sparse.spdiags(A.diagonal(), [0], n, n)
    ADinv = sparse.spdiags([1 / x for x in A.diagonal()], [0], n, n)
    Pinv = (AD - omega * AL) @ ADinv @ (AD - omega * AU) / (omega * (2
        - omega))

    r0 = b - A @ guess
    rho = np.linalg.norm(r0)
    r0 /= rho

    Q = np.zeros((n, maxIterations))
    Q[:, 0] = r0
    H = sparse.lil_matrix((maxIterations + 1, maxIterations))
    residuals = []
    for iteration in range(maxIterations):
        z = sparse.linalg.spsolve(Pinv, Q[:, iteration])
        v = A @ z
        for j in range(iteration + 1):
            H[j, iteration] = np.dot(Q[:, j], v)
            v -= H[j, iteration] * Q[:, j]

        norm_v = np.linalg.norm(v)
        H[iteration + 1, iteration] = norm_v
        Q[:, iteration + 1] = v / norm_v

        e = np.zeros(maxIterations + 1)
        e[0] = rho
        y, *_ = linalg.lstsq(H.toarray(), e)
        residual = np.linalg.norm(e - H @ y) / rho
        residuals.append(residual)

        if residual < tolerance or iteration == n - 1:
            w = sparse.linalg.spsolve(Pinv, Q @ y)
            x = guess + w
            break

    return x, residuals, iteration + 1
```

Third, the code from `myCG_SSOR.py`:

```python
import numpy as np
from scipy import sparse
from scipy.sparse import linalg


def myCG_SSOR(A, guess, b, tolerance=1e-12, maxIterations=1000):
    n = A.shape[0]
    maxIterations = min(n, maxIterations)
    omega = 1.9
    AL = -1 * sparse.tril(A, k=-1)
    AU = -1 * sparse.triu(A, k=1)
    AD = sparse.spdiags(A.diagonal(), [0], n, n)
    ADinv = sparse.spdiags([1 / x for x in A.diagonal()], [0], n, n)
    Pinv = (AD - omega * AL) @ ADinv @ (AD - omega * AU) / (omega * (2
        - omega))

    Rvecs = np.zeros((n, maxIterations))
    Qvecs = np.zeros((n, maxIterations))
    Pvecs = np.zeros((n, maxIterations))
    Xvecs = np.zeros((n, maxIterations))
    Xvecs[:, 0] = guess

    r0 = b - A @ guess
    q0 = linalg.spsolve(Pinv, r0)
    p0 = np.copy(q0)
    Rvecs[:, 0], Qvecs[:, 0], Pvecs[:, 0] = r0, q0, p0

    residuals = [np.linalg.norm(r0)]
    for k in range(1, maxIterations):
        Aonp = A @ Pvecs[:, k - 1]
        alpha = np.dot(Rvecs[:, k - 1], Qvecs[:, k - 1]) / np.dot(Pvecs
            [:, k - 1], Aonp)

        xk = Xvecs[:, k - 1] + alpha * Pvecs[:, k - 1]
        rk = Rvecs[:, k - 1] - alpha * Aonp

        residual = np.linalg.norm(rk) / residuals[0]
        residuals.append(residual)
        if residual < tolerance or k == maxIterations - 1:
            return xk, residuals, k

        Xvecs[:, k] = xk
        Rvecs[:, k] = rk

        qk = sparse.linalg.spsolve(Pinv, rk)
        Qvecs[:, k] = qk

        beta = np.dot(rk, qk) / np.dot(Rvecs[:, k - 1], Qvecs[:, k -
            1])
        Pvecs[:, k] = qk + beta * Pvecs[:, k - 1]
    return xk, residuals, maxIterations
```

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

```
import numpy as np
from scipy import sparse


def myCG(A, guess, b, tolerance=1e-12, maxIterations=1000):
    n = A.shape[0]
    maxIterations = min(n, maxIterations)

    Rvecs = np.zeros((n, maxIterations))
    Pvecs = np.zeros((n, maxIterations))
    Xvecs = np.zeros((n, maxIterations))
    Xvecs[:, 0] = guess

    r0 = b - A @ guess
    p0 = np.copy(r0)
    Rvecs[:, 0], Pvecs[:, 0] = r0, p0

    residuals = [np.linalg.norm(r0)]
    for k in range(1, maxIterations):
        Aonp = A @ Pvecs[:, k - 1]
        alpha = np.dot(Rvecs[:, k - 1], Rvecs[:, k - 1]) / np.dot(Pvecs
            [:, k - 1], Aonp)

        xk = Xvecs[:, k - 1] + alpha * Pvecs[:, k - 1]
        rk = Rvecs[:, k - 1] - alpha * Aonp

        residual = np.linalg.norm(rk) / residuals[0]
        residuals.append(residual)
        if residual < tolerance or k == maxIterations - 1:
            return xk, residuals, k

        Xvecs[:, k] = xk
        Rvecs[:, k] = rk

        beta = np.dot(rk, rk) / np.dot(Rvecs[:, k - 1], Rvecs[:, k -
            1])
        Pvecs[:, k] = rk + beta * Pvecs[:, k - 1]
    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`:

```python
import numpy as np
import sys

sys.path.insert(0, "../assignment1/problemThree")
from build_laplace_2d import build_laplace_2D
from myGMRES import myGMRES
from myGMRESSSOR import myGMRES_SSOR
from myCG import myCG
from myCGSSOR import myCG_SSOR

print("**********")

maxIterations = 500
tolerance = 1e-12
N = 8
n = N ** 2
A = build_laplace_2D(N)
x_exact = np.random.rand(n)
b = A @ x_exact
x0 = np.zeros(n)

x_gmres, resgmres, stepsgmres = myGMRES(A, x0, b, tolerance,
    maxIterations)
x_gmresSSOR, resgmresSSOR, stepsgmresSSOR = myGMRES_SSOR(
    A, x0, b, tolerance, maxIterations
)
x_cg, rescg, stepscg = myCG(A, x0, b, tolerance, maxIterations)
x_cgSSOR, rescgSSOR, stepscgSSOR = myCG_SSOR(A, x0, b, tolerance,
    maxIterations)

print("error for GMRES, # of steps")
error = np.linalg.norm(x_exact - x_gmres)
print(f"error: {error}")
print(f"steps: {stepsgmres}")

print("----")

print("error for GMRES_SSOR, # of steps")
error = np.linalg.norm(x_exact - x_gmresSSOR)
print(f"error: {error}")
print(f"steps: {stepsgmresSSOR}")

print("----")

print("error for CG, # of steps")
error = np.linalg.norm(x_exact - x_cg)
print(f"error: {error}")
print(f"steps: {stepscg}")

print("----")

print("error for CG_SSOR, # of steps")
```

```python
error = np.linalg.norm(x_exact - x_cgSSOR)
print(f"error: {error}")
print(f"steps: {stepscgSSOR}")
```

(e)

Now here's the code for `driverPCG_PGMRES.py`:

```python
import numpy as np
import sys

sys.path.insert(0, "../assignment1/problemThree")

from build_laplace_2d import build_laplace_2D
from myGMRES import myGMRES
from myGMRESSSOR import myGMRES_SSOR
from myCG import myCG
from myCGSSOR import myCG_SSOR

import matplotlib.pyplot as plt

plt.style.use("ggplot")

maxIterations = 400
tolerance = 1e-10

N = 32
n = N ** 2
A = build_laplace_2D(N)

b = np.ones(n)

x0 = np.zeros(n)

x_gmres, resgmres, stepsgmres = myGMRES(A, x0, b, tolerance,
    maxIterations)
x_gmresSSOR, resgmresSSOR, stepsgmresSSOR = myGMRES_SSOR(
    A, x0, b, tolerance, maxIterations
)
x_cg, rescg, stepscg = myCG(A, x0, b, tolerance, maxIterations)
x_cgSSOR, rescgSSOR, stepscgSSOR = myCG_SSOR(A, x0, b, tolerance,
    maxIterations)

print(f"Steps for GMRES:           {stepsgmres}")
print("----")
print(f"Steps for GMRES with SSOR: {stepsgmresSSOR}")
print("----")
print(f"Steps for CG:              {stepscg}")
print("----")
print(f"Steps for CG with SSOR:    {stepscgSSOR}")

fig = plt.figure()
ax = plt.subplot(111)
ax.plot(resgmres, label="GMRES")
ax.plot(resgmresSSOR, label="GMRES with SSOR")
ax.plot(rescg, label="CG")
ax.plot(rescgSSOR, label="CG with SSOR")
```

```

plt.title("Residual Size")
ax.legend()
ax.set_xlabel(r"Iterations $n$")
ax.set_ylabel(r"Residual $\|\|r_n\|\|$")
plt.yscale("log")
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-

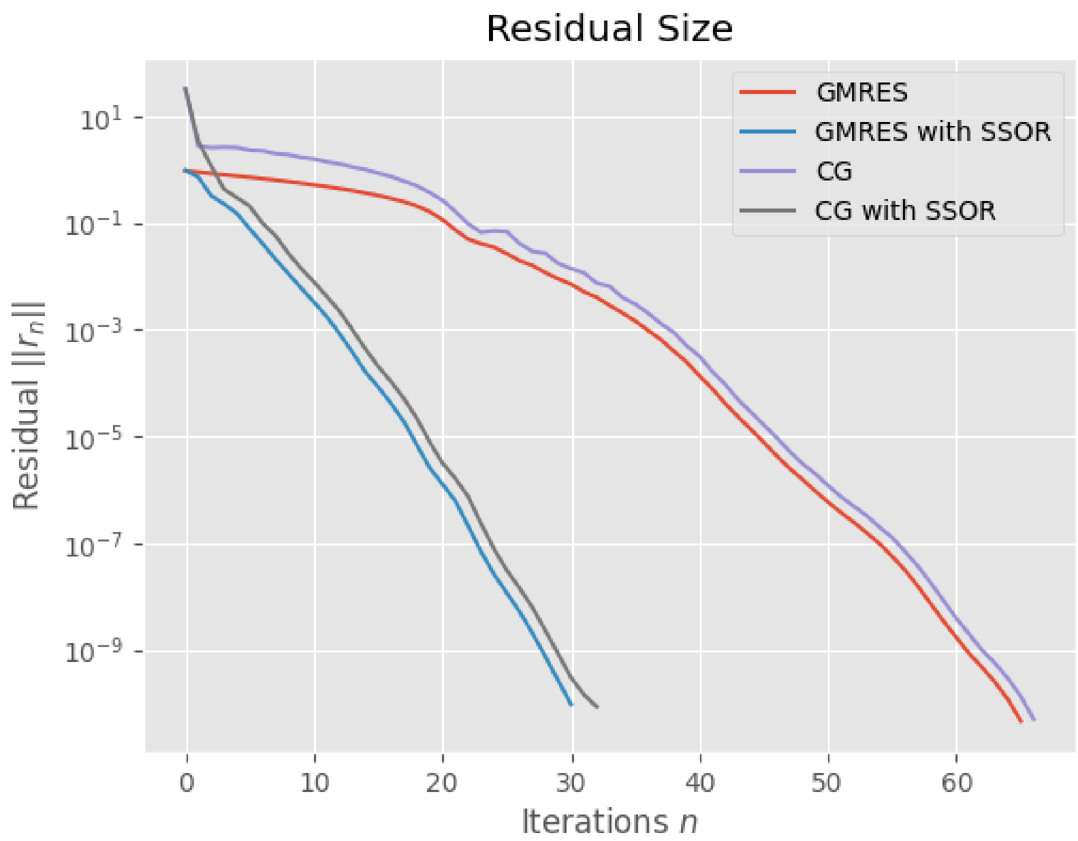

Figure 4.1: Residual Size as a function of iteration number

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`:

```
import numpy as np
import sys

sys.path.insert(0, "../assignment1/problemThree")
from build_laplace_2d import build_laplace_2D
from myGMRES import myGMRES
from myGMRESSSOR import myGMRES_SSOR
from myCG import myCG
from myCGSSOR import myCG_SSOR
import matplotlib.pyplot as plt

plt.style.use("ggplot")

Ns = range(80, 150, 10)
ns = [N ** 2 for N in Ns]
tol = 1e-10

steps = {"gmres": [], "gmresssor": [], "cg": [], "cgssor": []}
for N in Ns:
    print(N)
    n = N ** 2
    A = build_laplace_2D(N)
    b = np.ones(n)
    x0 = np.zeros(n)

    *_, stepsgmres = myGMRES(A, x0, b, tol)
    print("\tdone with GMRES")
    *_, stepsgmresSSOR = myGMRES_SSOR(A, x0, b, tol)
    print("\tdone with GMRES SSOR")
    *_, stepscg = myCG(A, x0, b, tol)
    print("\tdone with CG")
    *_, stepscgSSOR = myCG_SSOR(A, x0, b, tol)
    print("\tdone with CG SSOR")

    steps["gmres"].append(stepsgmres)
    steps["gmresssor"].append(stepsgmresSSOR)
    steps["cg"].append(stepscg)
    steps["cgssor"].append(stepscgSSOR)

fig = plt.figure()
ax = plt.subplot(111)
x = np.log(ns)
ax.plot(x, np.log(steps["gmres"]), label="GMRES")
ax.plot(x, np.log(steps["gmresssor"]), label="GMRES with SSOR")
ax.plot(x, np.log(steps["cg"]), label="CG")
ax.plot(x, np.log(steps["cgssor"]), label="CG with SSOR")

ax.legend()
ax.set_xlabel(r"Log of Problem Size")
ax.set_ylabel(r"Log of # of Iterations")
plt.show()

slope_gmres, _ = np.polyfit(x, np.log(steps["gmres"]), 1)
print(f"GMRES loglog slope: {slope_gmres}")
slope_cg, _ = np.polyfit(x, np.log(steps["cg"]), 1)
```

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

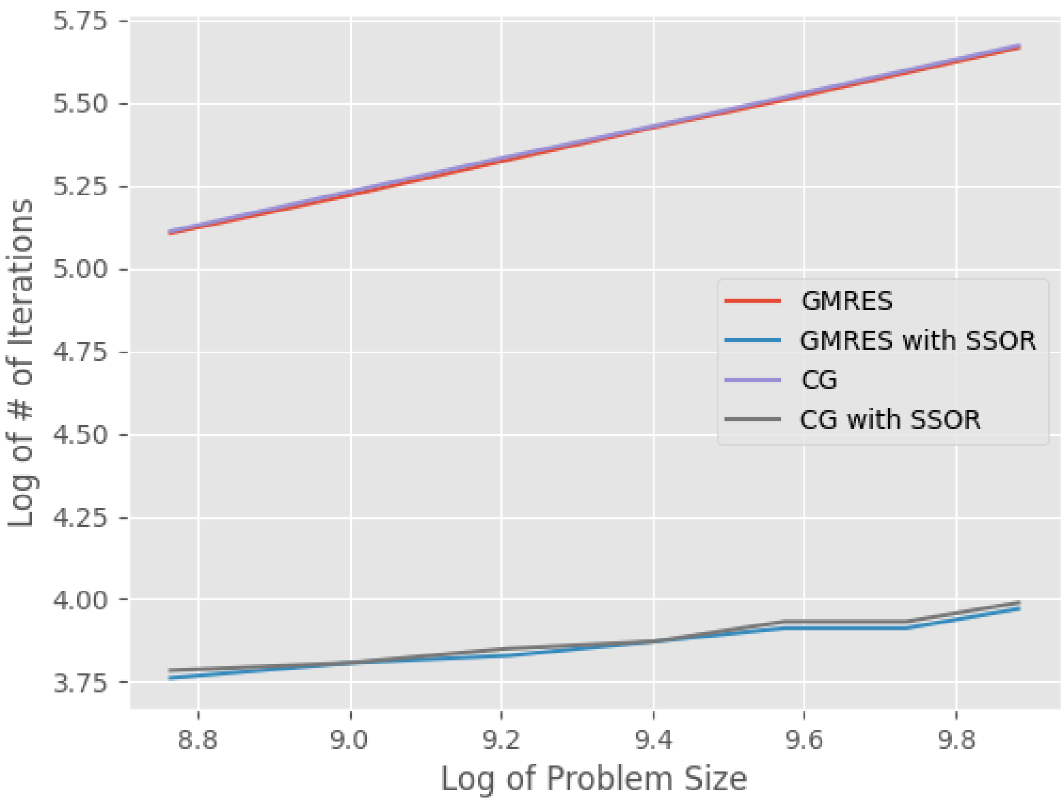

Figure 4.2: Log-log plot

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!