5주차 과제 #7

2022145079 임혜린

본 과제는 Python, VS Code를 사용하였음을 밝힙니다.

공통 조건 (Poisson equations)

Poisson equations

2차원 Poisson equation은 다음과 같이 주어진다.

$$\nabla^2 u(x,y) = f(x,y)\ for\ (x,y) \in \Omega$$

그리고 경계 $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$ 는 다음과 같이 주어진다.

$$u(x,y) = g(x,y)\ on\ \partial\Omega_D\ and\ \partial u/\partial n = h(x,y)\ on\ \partial\Omega_N$$

n은 경계에 대한 수직방향이며, $\partial\Omega_D$는 Dirichlet 경계를, $\partial\Omega_N$는 Neumann 경계를 의미한다.

푸아송 방정식은 라플라스 방정식을 일반화한 2차 편미분 방정식이다. 유체역학의 영역에서는 압력, 속도 퍼텐셜, 스트림 함수 등에 자주 쓰인다. 압력의 경우에는, 비압축성 유동에서 Navier-Stokes 방정식과 연속방정식을 이용하여 압력에 대한 직접적인 방정식을 만들면 푸아송 방정식 형태가 유도된다.

$$\nabla^2 p = -\rho\,\nabla\cdot[(\mathbf{u}\cdot\nabla)\mathbf{u}] + \rho\,\nabla\cdot\mathbf{f}$$

속도 퍼텐셜의 경우에는, 비회전성 유동에서 source나 sink가 존재한다면 라플라스 방정식이 아닌 푸아송 방정식의 형태로 나타난다.

$$\nabla^2\phi = -\sigma$$

스트림 함수와 와류의 관계 또한 푸아송 방정식이 성립한다.

$$\nabla^2\psi = -\omega$$

이렇듯 유체역학에 빈번히 적용되는 푸아송 방정식을 CFD에 적용하는 연습을 할 예정이다.

1-1

1. (Iterative Poisson solver) 정사각 계산영역 $[0,1] \times [0,1]$이 주어졌고, 경계에서의 u=0이며, $f(x,y) = \sin(\pi x)\sin(\pi y)$로 주어졌을 때, 포아송 방정식을 계산하시오

(1) 반복계산을 사용하여 Poisson equation 을 균일 격자계에서 주변 5개 격자점을 사용하여 계산하시오. 1) Jacobi method, 2) Gauss-Seidel method, 3) Gauss-Seidel method with successive over-relaxation (SOR)

3가지 방법 모두 기본적으로 아래와 같은 환경에서 코딩하였다.

```
import numpy as np
import matplotlib.pyplot as plt
import time

n=41

x_list=np.linspace(0, 1, n)
y_list=np.linspace(0, 1, n)
X,Y=np.meshgrid(x_list, y_list)
h=x_list[1]-x_list[0]

u0=np.zeros((n, n))

def f(x, y):
    return np.sin(np.pi*x)*np.sin(np.pi*y)
```

1) Jacobi method

주변 변수를 전부 이전 회차의 값을 이용해 미지수를 계산하는 방식이다. 식은 아래와 같다.

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - h^2 f_{i,j}\right)$$

0으로 채워진 (n, n) 2차원 벡터 u_new를 만든 후, 이전 값들을 모은 u_list_J의 가장 최근 값, 즉 직전 값 5개를 이용하여 u_new의 값을 채워나간다.

```
u_new[i, j]=0.25*(u_list_J[k][i+1,j]+u_list_J[k][i-
1,j]+u_list_J[k][i,j+1]+u_list_J[k][i,j-1]-(h**2)*f(x_list[i],y_list[j]))
```

모두 채워진 후 이 u_new를 u_list_J에 복사해 추가함으로써 모든 반복 횟수 k에서 u_list_J[k]가 직전 값이 되도록 한다. 즉, u_new의 전체가 채워지기 전에는 늘 직전 값만이 이용되는 것이다.

반복은 직전 값과 새로운 값의 L2 값이 10^{-5}보다 작아지면 멈춘다.

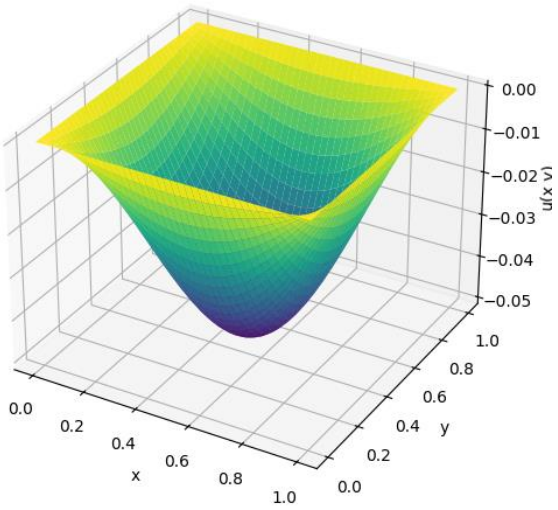
1-2번 문제에 사용할 computational time을 측정하기 위해 코드 위 아래에 start_time과 end_time을 두어 computational time을 comp_time_J에 저장하였다.

반복횟수 k-1은 k_J로 저장된다.

```
start_time=time.time()

u_list_J=[u0,]

k=0
while k<2000:
    u_new=np.zeros((n, n))
    for i in range(1, n-1):
        for j in range(1, n-1):
            u_new[i, j]=0.25*(u_list_J[k][i+1,j]+u_list_J[k][i-1,j]+u_list_J[k][i
    k+=1
    if np.linalg.norm(u_new-u_list_J[-1])<1e-5:
        break
    u_list_J.append(u_new.copy())


fig = plt.figure(figsize=(8, 6))
ax=fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,u_list_J[-1], cmap='viridis')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u(x,y)')
ax.set_title('Poisson Equation Solution by Jacobi Method')
plt.show()

# 반복 횟수 도출
print(f"반복 횟수: {k-1}")
k_J=k-1

# computational time 측정
end_time=time.time()
comp_time_J=end_time-start_time
print(comp_time_J)
```

Poisson Equation Solution by Jacobi Method

결과




```
반복 횟수: 1861
28.466504335403442
```

2) Gauss-Seidel method

한 회차에서는 새롭게 갱신된 값이 있어도 모두 이전 회차의 값을 사용하는 Jacobi method와는 달리, Gauss-Seidel method는 새롭게 갱신된 값을 즉시 다른 변수 계산에 적용한다. 따라서 Jacobi method보다 빠르게 수렴하는 경향을 보인다. 식은 아래와 같다.

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i+1,j}^{(k)} + \quad /* \text{아래쪽: 이전값} */ u_{i-1,j}^{(k+1)} + \quad /* \text{위쪽: 최신값} */ u_{i,j+1}^{(k)} + \quad / \right.$$

$$\left. * \text{오른쪽: 이전값} */ u_{i,j-1}^{(k+1)} \quad /* \text{왼쪽: 최신값} */ - h^2 f_{i,j}\right)$$

새로운 값이 바로 반영되어야 하기 때문에 좌변과 우변 모두 같은 2차원 벡터를 사용하였다. 새로운 값이 즉시 u에 덮어써지기 때문에 [i-1, j], [i, j-1]이 새로운 값으로 대응된다.

```
u[i, j]=0.25*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-
(h**2)*f(x_list[i],y_list[j]))
```

반복은 Jacobi와 마찬가지로 직전 값과 새로운 값의 L2 값이 10^{-5}보다 작아지면 멈춘다.

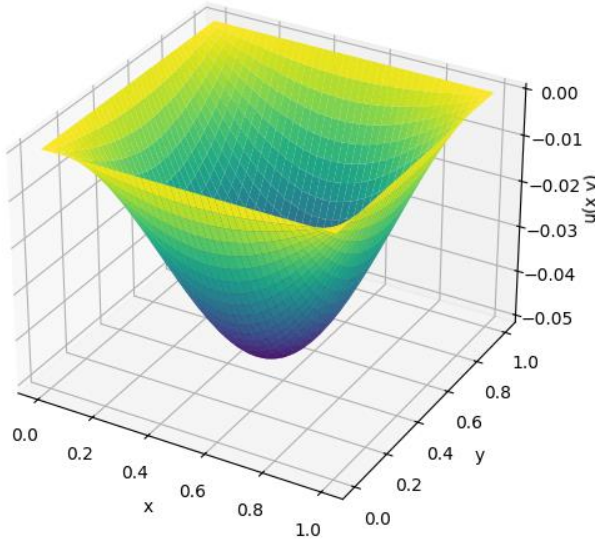
1-2번 문제에 사용할 computational time을 측정하기 위해 코드 위 아래에 start_time과 end_time을 두어 computational time을 comp_time_G에 저장하였다.

반복횟수 k-1은 k_G로 저장된다.

```
start_time=time.time()

u_list_G=[u0,]

k=0
u=np.zeros((n, n))
while k<2000:
    for i in range(1, n-1):
        for j in range(1, n-1):
            u[i, j]=0.25*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-(h**2)*f(x_list[i]
    k+=1
    if np.linalg.norm(u-u_list_G[-1])<1e-5:
        break
    u_list_G.append(u.copy())

fig = plt.figure(figsize=(8, 6))
ax=fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,u_list_G[-1], cmap='viridis')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u(x,y)')
ax.set_title('Poisson Equation Solution by Gauss-Seidel Method')
plt.show()
```

```
# 반복 횟수 도출
print(f"반복 횟수: {k-1}")
k_G=k-1

# computational time 측정
end_time=time.time()
comp_time_G=end_time-start_time
print(comp_time_G)
```

Poisson Equation Solution by Gauss-Seidel Method 결과




```
반복 횟수: 1043
20.246073722839355
```

3) Gauss-Seidel + SOR (Successive Over-Relaxation)

기본적으로 Gauss-Seidel method를 사용하되, 수렴 속도 증가를 위해 해의 새 값에 완화계수를 곱하여 적절히 보정하는 방식이다. 완화계수가 너무 크거나 작으면 발산하기 때문에 적절히 조절하는 것이 중요하다.

$$u_{i,j}^{(k+1)} = (1-\omega)u_{i,j}^{(k)} + \frac{\omega}{4}\left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)} - h^2 f_{i,j}\right)$$

코드의 틀은 Gauss-Seidel과 같으나 완화계수 ω를 넣어 더욱 빠르게 수렴하도록 하였다.

```
u[i, j]=(1-w)*u[i, j]+0.25*w*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-
(h**2)*f(x_list[i],y_list[j]))
```

반복은 마찬가지로 직전 값과 새로운 값의 L2 값이 10^{-5}보다 작아지면 멈춘다.

1-2번 문제에 사용할 computational time을 측정하기 위해 코드 위 아래에 start_time과 end_time을 두어 computational time을 comp_time_GSOR에 저장하였다.

반복횟수 k-1은 k_GSOR로 저장된다.

수렴을 보장하기 위해서는 |1-ω|<1이어야 한다. 따라서 ω는 0<ω<2이어야 하고, 수렴 속도를 줄이기 위해서는 1<ω<2가 요구된다. 따라서 이 범위에서의 가장 최적의 ω를 찾기 위하여 1.1부터 1.9까지 0.1 간격으로 ω를 적용하였을 때의 반복 횟수와 소요 시간을 살펴보았다.

```
# Gauss-Seidel method with successive over-relaxation (SOR)
# 최적의 완화계수 w 찾기

u0=np.zeros((n, n))
u_list_GSOR=[u0,]

for w in [1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]:
    start_time=time.time()
    k=0
    u=np.zeros((n, n))
    while k<1000:
        for i in range(1, n-1):
            for j in range(1, n-1):
                u[i, j]=(1-w)*u[i, j]+0.25*w*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]
        k+=1
        if np.linalg.norm(u-u_list_GSOR[-1])<1e-5:
            break
        u_list_GSOR.append(u.copy())
    # 반복 횟수 도출
    print(f"w={w}일 때 반복 횟수: {k-1}")
    k_GSOR=k-1

    # computational time 측정
    end_time=time.time()
    comp_time_GSOR=end_time-start_time
    print(f'소요 시간:{comp_time_GSOR}')
    print()
```

결과

```
w=1.1일 때 반복 횟수: 880
소요 시간:13.029031753540039

w=1.2일 때 반복 횟수: 739
소요 시간:11.784751653671265

w=1.3일 때 반복 횟수: 614
소요 시간:8.293700933456421

w=1.4일 때 반복 횟수: 504
소요 시간:6.267594575881958

w=1.5일 때 반복 횟수: 404
소요 시간:4.721038818359375
```

```
w=1.6일 때 반복 횟수: 312
소요 시간:3.634824514389038

w=1.7일 때 반복 횟수: 226
소요 시간:2.846325159072876

w=1.8일 때 반복 횟수: 141
소요 시간:1.7256946563720703

w=1.9일 때 반복 횟수: 93
소요 시간:1.2337543964385986
```

1.9일 때 가장 반복 횟수와 소요 시간이 작았다.

조금 더 자세히 알아보기 위해 1.85부터 1.95까지 0.1 간격으로 다시 측정하였다.

```
# Gauss-Seidel method with successive over-relaxation (SOR)
# 최적의 완화계수 w 찾기

u0=np.zeros((n, n))
u_list_GSOR=[u0,]

for w in [1.85, 1.86, 1.87, 1.88, 1.89, 1.9, 1.91, 1.92, 1.93, 1.94, 1.95]:
    start_time=time.time()
    k=0
    u=np.zeros((n, n))
    while k<500:
        for i in range(1, n-1):
            for j in range(1, n-1):
                u[i, j]=(1-w)*u[i, j]+0.25*w*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]
        k+=1
        if np.linalg.norm(u-u_list_GSOR[-1])<1e-5:
            break
        u_list_GSOR.append(u.copy())
    # 반복 횟수 도출
    print(f"w={w}일 때 반복 횟수: {k-1}")
    k_GSOR=k-1

    # computational time 측정
    end_time=time.time()
    comp_time_GSOR=end_time-start_time
    print(f'소요 시간:{comp_time_GSOR}')
    print()
```

결과

```
w=1.85일 때 반복 횟수: 90
소요 시간:1.2321979999542236

w=1.86일 때 반복 횟수: 77
소요 시간:1.0579793453216553

w=1.87일 때 반복 횟수: 80
소요 시간:0.933905839920044

w=1.88일 때 반복 횟수: 81
소요 시간:1.3920783996582031

w=1.89일 때 반복 횟수: 89
소요 시간:1.5827629566192627
```

```
w=1.9일 때 반복 횟수: 93
소요 시간:1.1985352039337158

w=1.91일 때 반복 횟수: 106
소요 시간:1.3341445922851562

w=1.92일 때 반복 횟수: 120
소요 시간:1.8770923614501953

w=1.93일 때 반복 횟수: 133
...

w=1.95일 때 반복 횟수: 190
소요 시간:3.5369129180908203
```

ω=1.86일 때 가장 작은 반복 횟수와 소요 시간을 보였으므로 완화계수 ω는 1.86으로 채택하였다.

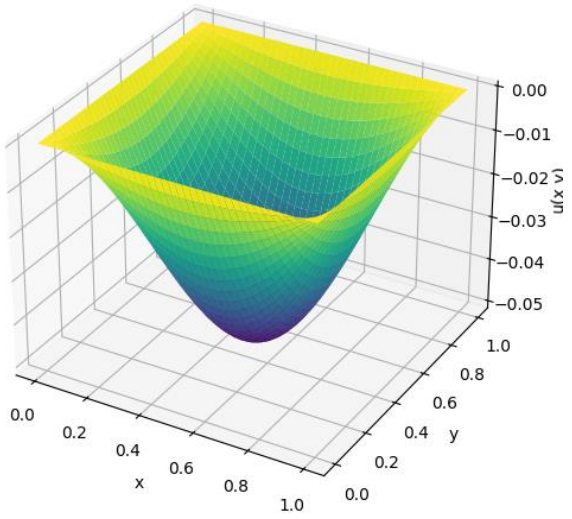
또한, 2-2번 문제에서 f의 종류에 따른 u의 단면 모양의 차이를 보여주기 위하여 u의 중앙을 지나는 단면의 값 (x=0.5, 즉 u[21]일 때의 u값)을 u_middle_f1로 따로 저장해주었다. 저장된 u_middle_f1은 아래와 같다.

```
[ 0.         -0.00396455 -0.00790478 -0.0117964  -0.01561523 -0.01933775
 -0.02294102 -0.02640284 -0.02970189 -0.03281781 -0.03573139 -0.03842466
 -0.04088102 -0.04308533 -0.04502399 -0.04668505 -0.04805828 -0.04913521
 -0.0499092  -0.05037549 -0.0505312  -0.05037538 -0.04990898 -0.04913489
 -0.04805788 -0.04668458 -0.04502346 -0.04308476 -0.04088043 -0.03842406
 -0.03573079 -0.03281723 -0.02970135 -0.02640235 -0.02294056 -0.01933735
 -0.01561491 -0.01179621 -0.00790477 -0.00396461  0.        ]
```

```python
start_time=time.time()

u_list_GSOR=[u0,]

k=0
w=1.86
u=np.zeros((n, n))
while k<2000:
    for i in range(1, n-1):
        for j in range(1, n-1):
            u[i, j]=(1-w)*u[i, j]+0.25*w*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-(h*
    k+=1
    if np.linalg.norm(u-u_list_GSOR[-1])<1e-5:
        break
    u_list_GSOR.append(u.copy())

# 2-2 문제에 쓸 단면 그래프를 위한 작업
u_middle_f1=u_list_GSOR[-1][int((n-1)/2)+1]
print(u_middle_f1)

fig = plt.figure(figsize=(8, 6))
ax=fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,u_list_GSOR[-1], cmap='viridis')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u(x,y)')
ax.set_title('Poisson Equation Solution by Gauss-Seidel Method with SOR')
plt.show()

# 반복 횟수 도출
print(f"반복 횟수: {k-1}")
k_GSOR=k-1
```

```python
# computational time 측정
end_time=time.time()
comp_time_GSOR=end_time-start_time
print(comp_time_GSOR)
```

Poisson Equation Solution by Gauss-Seidel Method with SOR 결과




```
반복 횟수: 77
2.293455123901367
```

(2) 각 반복계산 방법의 Performance를 보이시오. Ex) norm of residual, errors, computational time, etc

Number of repetitions, Norm of residual, Errors, Computational time을 순서대로 도출하였다.

Norm of residual (잔차 노름)은 수치해와 실제 해가 얼마나 다른 지, 즉 현재 식이 PDE를 얼마나 위반하고 있는지를 측정하는 값이다. 다시 말해 아래 공식의 좌변과 우변이 얼마나 다른 지를 나타내는 값이라고 할 수 있다.

$$\frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} = f(x_i, y_j)$$

이때 Norm of residual과 Error 모두 L2 norm을 사용하였다.

- Jacobi method

```python
## number of repetitions
print(f'Reps: {k_J}')

## Norm of residual
norm_J=np.zeros((n, n))
for i in range(1, n-1):
    for j in range(1, n-1):
        norm_J[i,j]=f(x_list[i],y_list[j])-(u_list_J[-1][i+1,j] + u_list_J[-1][i-
print(f'Norm of residual: {np.linalg.norm(norm_J):.7f}')

## errors
print(f'error: {np.linalg.norm(u_list_J[-1]-u_exact(X, Y)):.7f}')

## computational time
print(f'computational time: {comp_time_J:.3f}s')
```

```
Reps: 1861
Norm of residual: 0.0639298
error: 0.0027194
computational time: 28.467s
```
결과

- Gauss-Seidel method

```
## number of repetitions
print(f'Reps: {k_G}')

## Norm of residual
norm_G=np.zeros((n, n))
for i in range(1, n-1):
    for j in range(1, n-1):
        norm_G[i,j]=f(x_list[i],y_list[j])-(u_list_G[-1][i+1,j] + u_list_G[-1][i-
print(f'Norm of residual: {np.linalg.norm(norm_G):.7f}')

## errors
print(f'error: {np.linalg.norm(u_list_G[-1]-u_exact(X, Y),2):.7f}')

## computational time
print(f'computational time: {comp_time_G:.3f}s')
```

```
Reps: 1044
Norm of residual: 0.0321095
error: 0.0011020
computational time: 15.476s
```
결과

- Gauss-Seidel + SOR (Successive Over-Relaxation)

```
## number of repetitions
print(f'Reps: {k_GSOR}')

## Norm of residual
norm_GSOR=np.zeros((n, n))
for i in range(1, n-1):
    for j in range(1, n-1):
        norm_GSOR[i,j]=f(x_list[i],y_list[j])-(u_list_GSOR[-1][i+1,j] + u_list_GS
print(f'Norm of residual: {np.linalg.norm(norm_GSOR):.7f}')

## errors
print(f'error: {np.linalg.norm(u_list_GSOR[-1]-u_exact(X, Y),2):.7f}')

## computational time
print(f'computational time: {comp_time_GSOR:.3f}s')
```

```
Reps: 77
Norm of residual: 0.0179107
error: 0.0005396
computational time: 2.293s
```
결과

세 방법의 결과를 표로 나타내면 다음과 같다.

	Jacobi	Gauss-Seidel	Gauss-Seidel+SOR
Reps	1861	1044	77
Norm of residual	0.0639298	0.0321095	0.0179107
Error	0.0027194	0.0011020	0.0005396
Computational time(s)	28.467	15.476	2.293

모든 항목의 값이 Jacobi > Gauss-Seidel > Gauss-Seidel+SOR로 나타난 것을 발견할 수 있다.

따라서 Poisson equation을 푸는 데에 Gauss-Seidel+SOR이 가장 적절함을 알 수 있다.

2. (Lineaerity) 다음의 포아송 방정식을 고려하여 문제를 푸시오.

$$\nabla^2 u(x,y) = f_1(x,y) + f_2(x,y)\ for\ (x,y) \in \Omega$$

그리고 경계 $\partial\Omega$에서 $u(x,y) = 0$ 를 만족하며, 정사각 계산영역 $[0,1] \times [0,1]$에서 진행하시오.

(1) Poisson equation의 해 u(x,y)를 SOR 방법을 사용하여 구하시오. 우항의 forcing 함수는 다음과 같이 주어진다.

$$f_1(x,y) = \sin(\pi x)\sin(\pi y)$$
$$f_2(x,y) = \exp\left(-100.0((x-0.5)^2 + (y-0.5)^2)\right)$$

f1, f2, f1+f2를 다음과 같이 정의하였다.

```python
def f1(x,y):
    return np.sin(np.pi*x)*np.sin(np.pi*y)

def f2(x,y):
    return np.exp(-100.0*((x-0.5)**2+(y-0.5)**2))

def f12(x,y):
    return f1(x,y)+f2(x,y)
```

1-1번의 Gauss-Seidel method+SOR과 코드의 틀은 같다. f1만 사용하였던 1-1과는 달리 f1+f2의 형태인 f12가 대신 적용되었다.

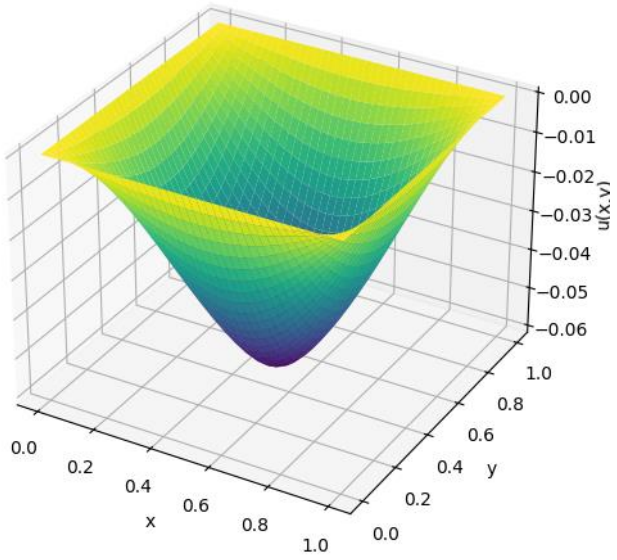
f는 완화 계수의 크기에 영향을 주지 않으므로 1-1번에서 도출하였던 ω=1.86을 채택하였다.

2-2번 문제에서 f의 종류에 따른 u의 단면 모양의 차이를 보여주기 위하여 u의 중앙을 지나는 단면의 값 (x=0.5, 즉 u[21]일 때의 u값)을 u_middle_f12로 따로 저장해주었다. 저장된 u_middle_f12은 아래와 같다.

```
[ 0.         -0.00429146 -0.0085614  -0.01278841 -0.01695133 -0.02102988
 -0.02500472 -0.02885777 -0.03257242 -0.03613395 -0.03952987 -0.04275014
 -0.04578681 -0.0486324  -0.05127584 -0.05369553 -0.05585072 -0.05767546
 -0.05908163 -0.05997512 -0.06028226 -0.059975   -0.05908138 -0.0576751
 -0.05585027 -0.053695   -0.05127525 -0.04863177 -0.04578615 -0.04274947
 -0.03952921 -0.03613331 -0.03257182 -0.02885722 -0.02500423 -0.02102943
 -0.01695097 -0.01278819 -0.00856138 -0.00429151  0.        ]
```

```
u_list_GSOR_f12=[u0,]

k=0
w=1.86
u=np.zeros((n, n))
while k<2000:
    for i in range(1, n-1):
        for j in range(1, n-1):
            u[i, j]=(1-w)*u[i, j]+0.25*w*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-(h*
    k+=1
    if np.linalg.norm(u-u_list_GSOR_f12[-1])<1e-5:
        break
    u_list_GSOR_f12.append(u.copy())

# 2-2 문제에 쓸 단면 그래프를 위한 작업
u_middle_f12=u_list_GSOR_f12[-1][int((n-1)/2)+1]
print(u_middle_f12)

# 반복 횟수 도출
print(f"반복 횟수: {k-1}")
k_GSOR_f12=k-1

fig = plt.figure(figsize=(8, 6))
ax=fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,u_list_GSOR_f12[-1], cmap='viridis')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u(x,y)')
ax.set_title('Poisson Equation Solution(f1+f2) by Gauss-Seidel with SOR')
plt.show()
```

Poisson Equation Solution(f1+f2) by Gauss-Seidel with SOR 결과



반복 횟수: 77

(2) 동일한 방법으로 f_2만을 고려한 포아송 방정식의 해 u_2를 구하시오

1-1번의 Gauss-Seidel method+SOR과 코드의 틀은 같다. 다만 f2가 대신 적용되었다.

위에서와 마찬가지로 완화 계수 ω=1.86를 적용하였다.

2-2번 문제에서 f의 종류에 따른 u의 단면 모양의 차이를 보여주기 위하여 u의 중앙을 지나는 단면의 값 (x=0.5, 즉 u[21]일 때의 u값)을 u_middle_f2로 따로 저장해주었다. 저장된 u_middle_f2은 아래와 같다.

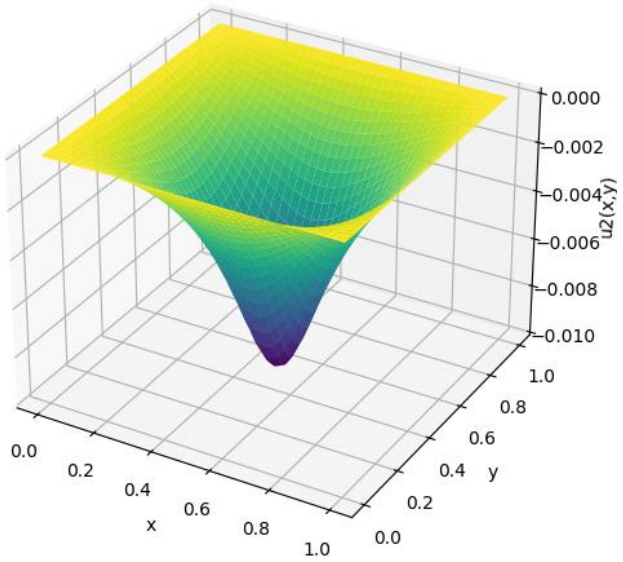
```
[ 0.         -0.00032638 -0.00065578 -0.00099082 -0.0013347  -0.00169059
 -0.00206209 -0.00245328 -0.00286889 -0.00331452 -0.0037969  -0.00432395
 -0.00490432 -0.00554566 -0.0062505  -0.00700919 -0.00779122 -0.00853911
 -0.00917138 -0.00959868 -0.00975022 -0.00959891 -0.00917182 -0.00853975
 -0.00779204 -0.00701016 -0.00625159 -0.00554685 -0.00490561 -0.00432533
 -0.00379836 -0.00331604 -0.00287045 -0.00245486 -0.00206365 -0.00169207
 -0.00133605 -0.00099198 -0.0006566  -0.0003269   0.        ]
```

```
u_list_GSOR_f2=[u0,]

k=0
w=1.86
u=np.zeros((n, n))
while k<2000:
    for i in range(1, n-1):
        for j in range(1, n-1):
            u[i, j]=(1-w)*u[i, j]+0.25*w*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1]-(h*
    k+=1
```

```
    if np.linalg.norm(u-u_list_GSOR_f2[-1])<1e-5:
        break
    u_list_GSOR_f2.append(u.copy())

# 2-2 문제에 쓸 단면 그래프를 위한 작업
u_middle_f2=u_list_GSOR_f2[-1][int((n-1)/2)+1]
print(u_middle_f2)

# 반복 횟수 도출
print(f"반복 횟수: {k-1}")
k_GSOR_f2=k-1

fig = plt.figure(figsize=(8, 6))
ax=fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,u_list_GSOR_f2[-1], cmap='viridis')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u2(x,y)')
ax.set_title('Poisson Equation Solution(f2) by Gauss-Seidel with SOR')
plt.show()
```

Poisson Equation Solution(f2) by Gauss-Seidel with SOR

결과




```
반복 횟수: 60
```

(3) 문제에서 구한 해 $u(x,y)$와 1. 문제에서 구한 $u_1(x,y)$ 그리고 2.-(2) 에서 구한 $u_2(x,y)$들의 해를 비교하고 이에 대한 생각을 서술하시오.

f1, f2, f1+f2의 Reps, Norm of residual을 구하였다. F2는 수학적 닫힌 해가 없으므로 exact solution을 이용하는 Error을 도출할 수 없다. 따라서 Error은 제외하였다. 또한 f가 다른 상황에서 reputational time을 비교하는 것이 유의미하지 않다고 판단하여 이 또한 제외하였다.

- f1

```
print("f1")
## number of repetitions
print(f'Reps: {k_GSOR}')

## Norm of residual
norm_GSOR=np.zeros((n, n))
for i in range(1, n-1):
    for j in range(1, n-1):
        norm_GSOR[i,j]=f(x_list[i],y_list[j])-(u_list_GSOR[-1][i+1,j] + u_list_GSO
print(f'Norm of residual: {np.linalg.norm(norm_GSOR):.7f}')

## errors
print(f'error: {np.linalg.norm(u_list_GSOR[-1]-u_exact(X, Y),2):.7f}')
```

- f2

```
print("""
f2""")
## number of repetitions
print(f'Reps: {k_GSOR_f2}')

## Norm of residual
norm_GSOR_f2=np.zeros((n, n))
for i in range(1, n-1):
    for j in range(1, n-1):
        norm_GSOR_f2[i,j]=f2(x_list[i],y_list[j])-(u_list_GSOR_f2[-1][i+1,j] + u_l
print(f'Norm of residual: {np.linalg.norm(norm_GSOR_f2):.7f}')
```

- f1+f2

```
print("""
f12""")
## number of repetitions
print(f'Reps: {k_GSOR_f12}')

## Norm of residual
norm_GSOR_f12=np.zeros((n, n))
for i in range(1, n-1):
    for j in range(1, n-1):
        norm_GSOR_f12[i,j]=f12(x_list[i],y_list[j])-(u_list_GSOR_f12[-1][i+1,j] +
print(f'Norm of residual: {np.linalg.norm(norm_GSOR_f12):.7f}')
```

```
f1
Reps: 77
Norm of residual: 0.0179107
error: 0.0005396

f2
Reps: 60
Norm of residual: 0.0093782

f12
Reps: 77
Norm of residual: 0.0192503
```

결과

	f1	f2	f1+f2
Reps	77	60	77
Norm of residual	0.0179107	0.0093782	0.0192503

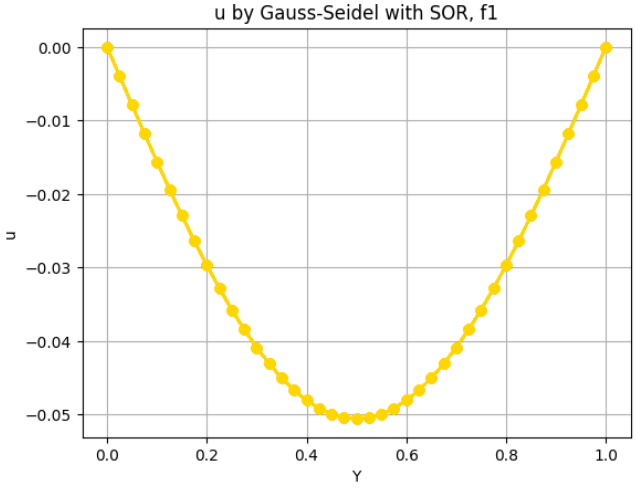
반복 횟수와 Norm of residual 모두 f1의 경우와 f1+f2의 경우가 근사하게 나타났다.

또한 3D 그래프의 단면을 직관적으로 비교해 보기 위해 단면 그래프를 그려보았다. 위의 코드에서 저장했던 u_middle_을 사용하였다.

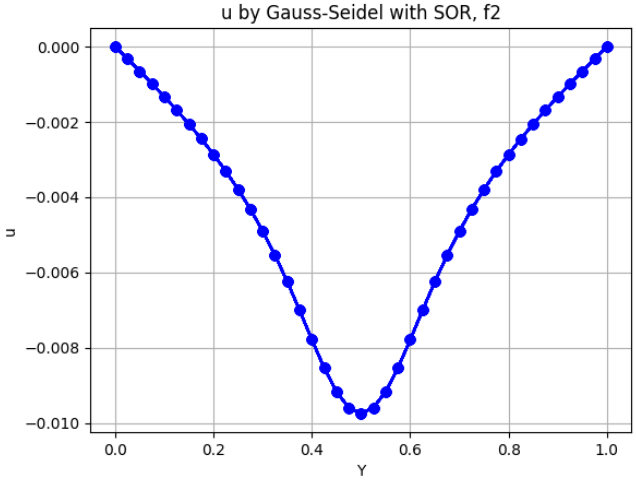
```
# f1
plt.plot(Y, u_middle_f1, marker='o', color='gold')
plt.xlabel('Y')
plt.ylabel('u')
plt.title('u by Gauss-Seidel with SOR, f1')
plt.grid()
plt.show()

# f2
plt.plot(Y, u_middle_f2, marker='o', color='b')
plt.xlabel('Y')
plt.ylabel('u')
plt.title('u by Gauss-Seidel with SOR, f2')
plt.grid()
plt.show()

# f1+f2
plt.plot(Y, u_middle_f12, marker='o', color='c')
plt.xlabel('Y')
plt.ylabel('u')
plt.title('u by Gauss-Seidel with SOR, f1+f2')
plt.grid()
plt.show()
```

결과

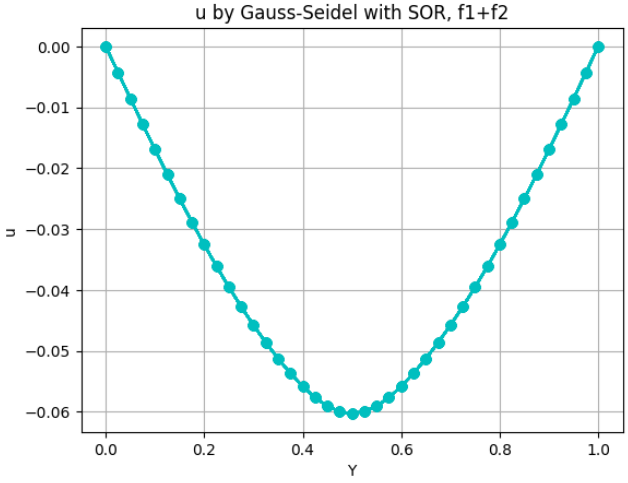
- f1


u by Gauss-Seidel with SOR, f1
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
u
0.0
0.2
0.4
0.6
0.8
1.0
Y

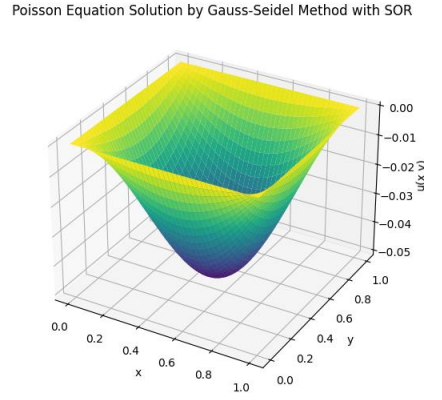
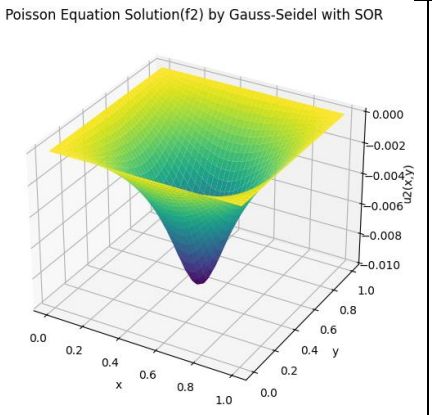
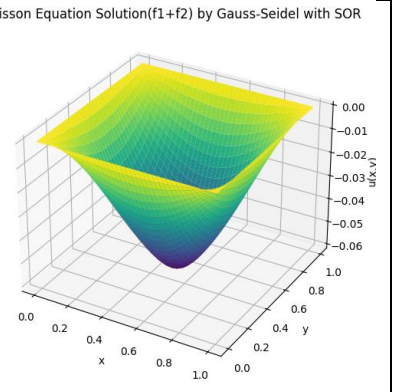
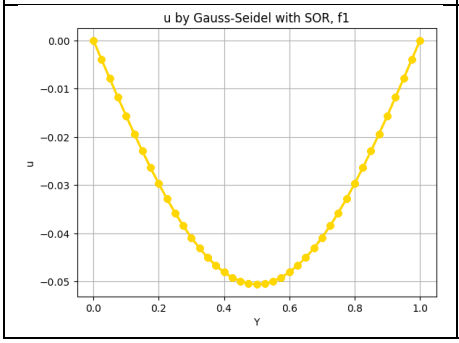
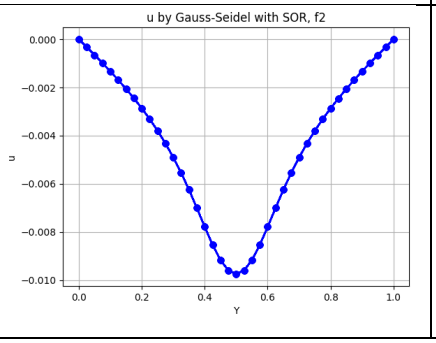
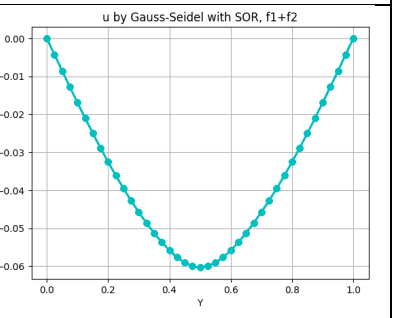
- f2


u by Gauss-Seidel with SOR, f2
0.000
−0.002
−0.004
−0.006
−0.008
−0.010
u
0.0
0.2
0.4
0.6
0.8
1.0
Y

- f1+f2


u by Gauss-Seidel with SOR, f1+f2
0.00
−0.01
−0.02
−0.03
−0.04
−0.05
−0.06
u
0.0
0.2
0.4
0.6
0.8
1.0
Y

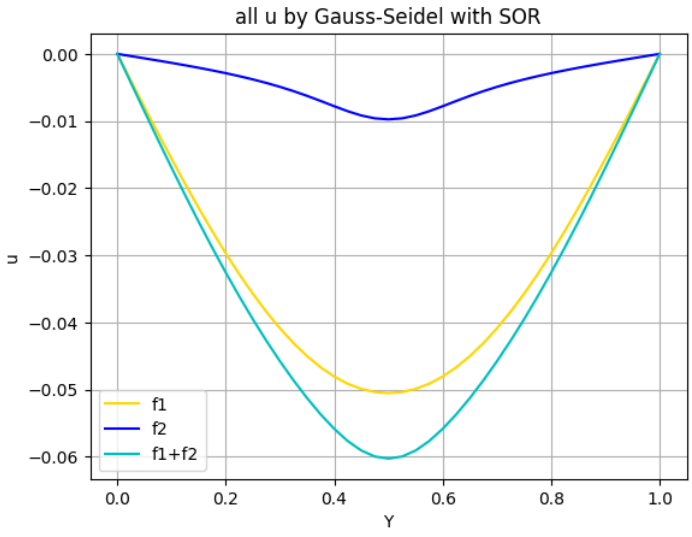
세 u의 3D 그래프와 단면 그래프를 비교하면 다음과 같다.

f1	f2	f1+f2
		
		

f1의 경우 2차 곡선과 비슷한 부드러운 곡선으로 나타난 반면, f2는 중앙에서 값이 급격히 작아져 좁고 우묵한 모양으로 나타났다. 두 f의 선형 조합인 f1+f2의 경우 해도 두 해가 더해진 모양으로 나타났다. 전체적인 모양은 f1와 더 유사하나 f1보다 더 좁고 날카로운 모양이다.

각 그래프의 깊이를 한 눈에 비교하기 위하여 단면 그래프 3가지를 겹쳐 보았다.

```
plt.plot(y_list, u_middle_f1, color='gold', label='f1')
plt.plot(y_list, u_middle_f2, color='b', label='f2')
plt.plot(y_list, u_middle_f12, color='c', label='f1+f2')
plt.xlabel('Y')
plt.ylabel('u')
plt.title('all u by Gauss-Seidel with SOR')
plt.grid()
plt.legend()
plt.show()
```



결과

f2에 의한 해가 f1에 비해서 전체적으로 약 5배 정도 작다. 이러한 요인 때문에 f1+f2의 모양이 f1에 가깝게 나온 것으로 보인다. 세 해를 도출하기 위한 반복 횟수와 Norm of residual가 f1와 f1+f2가 매우 비슷하게 나온 것 또한 이러한 해의 크기의 차이가 작용하였을 수도 있다.

+) 처음에는 완화 계수를 일방적으로 1.5를 넣어서 계산했었는데, 적절한 완화 계수 1.86을 찾은 후 이를 덮어쓰는 과정에서 1.5를 사용했을 때 Norm of residual과 error가 더 작게 나타남을 발견하였다. 이는 반복 횟수와 소요 시간을 가장 단축시키는 완화 계수가 정확도를 보장하지는 않는다는 것을 의미한다. 정확도가 중요한 상황에서는 이를 염두에 둘 필요가 있어 보인다.

(G+SOR, f1) <ω=1.5> <ω=1.86>

```
Reps: 404
Norm of residual: 0.0108045
error: 0.0000421
computational time: 6.193s
```

```
Reps: 77
Norm of residual: 0.0179107
error: 0.0005396
computational time: 2.293s
```

(G+SOR)

```
f1
Reps: 404
Norm of residual: 0.0108045
error: 0.0000421

f2
Reps: 291
Norm of residual: 0.0107229

f12
Reps: 410
Norm of residual: 0.0108046
```

```
f1
Reps: 77
Norm of residual: 0.0179107
error: 0.0005396

f2
Reps: 60
Norm of residual: 0.0093782

f12
Reps: 77
Norm of residual: 0.0192503
```