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.

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)

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

문제의 2차원 Poisson 방정식을 격자 점 (i, j)에 대해 2차 중앙 차분을 사용하면 다음과 같다.

$$-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} = h^2 f_{i,j}$$

위의 식은 선형 시스템으로, 다음과 같이 표현할 수 있다.

$$A\boldsymbol{u} = \boldsymbol{b}$$

Jacobi Method는 현재 시점의 이전 값을 이용해 새로운 값을 업데이트 하는 방법이다. 수식으로 표현하면 다음과 같다.

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

Gauss-Seidel Method는 계산한 최신 값을 바로 사용한다.

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

SOR Method는 Gauss-Seidel에 relaxation factor ω를 곱해 수렴 속도를 개선한 방법이다.

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

```python
#내부 격자 설정
n = 50
h = 1 / (n+1)
x = np.linspace(h, 1-h, n)
y = np.linspace(h, 1-h, n)
X, Y = np.meshgrid(x, y, indexing="ij")

# f(x,y)
f = np.sin(np.pi*X) * np.sin(np.pi*Y)
b = (h**2)*(f.flatten())
```

경계를 제외한 내부 격자를 50 X 50의 사이즈로 생성하고, 소스항 벡터 **b**를 생성했다.

Matrix A는 n^2 x n^2 크기의 sparse matrix로, n=3에 대해 다음과 같다.

$$A = \begin{bmatrix} B & I & 0 \\ I & B & I \\ 0 & I & B \end{bmatrix}, \qquad B = \begin{bmatrix} -4 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & -4 \end{bmatrix}$$

kronecker product를 통해 다음과 같이 A를 생성할 수 있다.

```python
# matrix A
T_1 = diags([np.ones(n-1),-4*np.ones(n),np.ones(n-1)], [-1, 0, 1], shape=(n,n))
T_2 = diags([np.ones(n-1), np.ones(n-1)], [-1, 1], shape=(n,n))
I = identity(n)
A = kron(I, T_1) + kron(T_2, I)
```

다음은 행렬을 이용하여 jacobi method를 구현한 코드이다.

```
def jacobi(A, b, tol=1e-6, max_iter=5000):
  D = diags(A.diagonal())
  R = A - D
  D_inv = diags(1/A.diagonal())
  x = np.zeros_like(b)
  for i in range(max_iter):
    x_new = D_inv @ (b - R @ x)
    if np.linalg.norm(x_new - x, ord=np.inf) < tol:
      return x_new, i, np.linalg.norm(x_new - x, ord=np.inf)
    x = x_new
  return x, max_iter
```

Tolerance와 최대 계산 횟수를 정해주었고, 행렬 A를 대각행렬 D와 나머지 R로 나누어 residual이 tolerance보다 작아지거나 최대 계산 횟수를 만족할 때까지 반복문을 실행해 수치적 해를 계산했다.

Gauss-Seidel method는 역행렬을 구하는 것이 어렵기 때문에 각 u값을 순차적으로 대입을 통해 구하는 코드를 작성했다.

```
def gauss(A, b, tol=1e-6, max_iter=5000):
  x = np.zeros_like(b)
  A = csr_matrix(A)
  for i in range(max_iter):
    x_new = np.copy(x)
    for j in range(A.shape[0]):
      row_start = A.indptr[j]
      row_end = A.indptr[j+1]
      Ai = A.indices[row_start:row_end]
      Av = A.data[row_start:row_end]
      x_new[j] = (b[j] - np.dot(Av, x_new[Ai]) + A[j,j]*x_new[j]) / A[j,j]
    if np.linalg.norm(x_new - x, ord=np.inf) < tol:
      return x_new, i, np.linalg.norm(x_new - x, ord=np.inf)
    x = x_new
  return x, max_iter
```

A를 csr형식으로 만들어 메모리의 효율성을 고려했다. 초기값을 u = [0 0… 0]으로 하여 $u_{1,1}$을 구하고, 이를 이용해 $u_{1,2}$를 구하고 결국 $u_{n,n}$까지 구하는 것을 반복하여 수치적 해를 구한다.

SOR은 Gauss-Seidel에 relaxation factor를 곱하기 때문에 다음과 같은 코드를 작성할 수 있다.

```
def SOR(A, b, omega=1.9, tol=1e-6, max_iter=5000):
  x = np.zeros_like(b)
  A = csr_matrix(A)
  for i in range(max_iter):
    x_new = np.copy(x)
    for j in range(A.shape[0]):
      row_start = A.indptr[j]
      row_end = A.indptr[j+1]
      Ai = A.indices[row_start:row_end]
      Av = A.data[row_start:row_end]
      x_new[j] = (1-omega)*x[j] +omega*(b[j] - np.dot(Av, x_new[Ai]) + A[j,j]*x_new[j])
    if np.linalg.norm(x_new - x, ord=np.inf) < tol:
      return x_new, i,  np.linalg.norm(x_new - x, ord=np.inf)
    x = x_new
  return x, max_iter
```

결과를 저장하고 출력하는 코드는 다음과 같다.

```
#결과 저장
u_j, it_j, norm_j = jacobi(A,b)
u_g, it_g, norm_g = gauss(A,b)
u_s, it_s, norm_s = SOR(A,b)

#계산횟수 출력
print(f"Jacobi: {it_j} iterations")
print(f"Gauss-Seidel: {it_g} iterations")
print(f"SOR: {it_s} iterations")

#norm of residual 계산
print(f"Jacobi norm of residual: {norm_j}")
print(f"Gauss-Seidel norm of residual: {norm_g}")
print(f"SOR norm of residual: {norm_s}")

#exact solution
u_exact = -(1/(2*np.pi**2))*(np.sin(np.pi*X)*np.sin(np.pi*Y)).flatten()

#error 계산
error_j = np.linalg.norm(u_j - u_exact)
error_g = np.linalg.norm(u_g - u_exact)
error_s = np.linalg.norm(u_s - u_exact)

print(f"Jacobi error: {error_j}")
print(f"Gauss-Seidel error: {error_g}")
print(f"SOR error: {error_s}")
```

결과는 다음과 같다.

```
Jacobi: 2405 iterations
Gauss-Seidel: 1386 iterations
SOR: 102 iterations
Jacobi norm of residual: 9.987911843770125e-07
Gauss-Seidel norm of residual: 9.974044652139025e-07
SOR norm of residual: 7.082424889779879e-07
Jacobi error: 0.013007034164258151
Gauss-Seidel error: 0.006275099570437602
SOR error: 0.00039673088601686633
```

계산횟수, norm of residual, error면에서 Jacobi, Gauss-Seidel, SOR 순으로 성능이 좋다는 점을 확인할 수 있다. 하지만 SOR은 적절한 relaxation factor를 사용해야 한다는 제약 조건이 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.

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

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

(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.

1번에서 소스항에 변화가 생겼으므로, 다음과 같이 소스항 벡터 **b**를 정의한다.

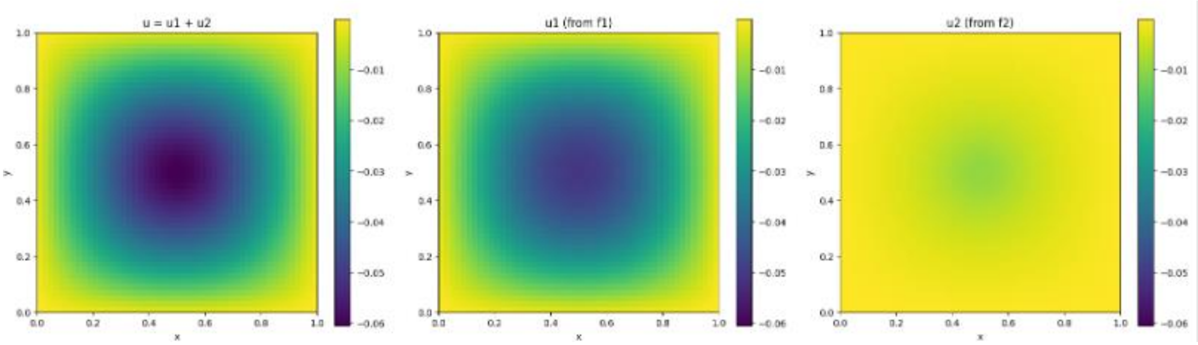
```python
# f(x,y)
f1 = np.sin(np.pi*X)*np.sin(np.pi*Y)
f2 = np.exp(-100*((X-0.5)**2+(Y-0.5)**2))
f = f1 + f2
b1 = (h**2)*(f1.flatten())
b2 = (h**2)*(f2.flatten())
b = (h**2)*(f.flatten())
```

결과를 저장하고 시각화 하는 코드는 다음과 같다. 시각화는 imshow를 이용했다.

```python
#결과 저장
u = SOR(A,b)
u1 = SOR(A, b1)
u2 = SOR(A, b2)

# 공통 vmin, vmax 계산
u_all = [u, u1, u2]
umin = min([np.min(ui) for ui in u_all])
umax = max([np.max(ui) for ui in u_all])
```

```python
fig, axs = plt.subplots(1, 3, figsize=(18, 5))

# u
im0 = axs[0].imshow(u.reshape(n, n), origin='lower', extent=[0,1,0,1],
                    cmap='viridis', vmin=umin, vmax=umax)
axs[0].set_title('u = u1 + u2')
axs[0].set_xlabel('x')
axs[0].set_ylabel('y')
plt.colorbar(im0, ax=axs[0])

# u1
im1 = axs[1].imshow(u1.reshape(n, n), origin='lower', extent=[0,1,0,1],
                    cmap='viridis', vmin=umin, vmax=umax)
axs[1].set_title('u1 (from f1)')
axs[1].set_xlabel('x')
axs[1].set_ylabel('y')
plt.colorbar(im1, ax=axs[1])

# u2
im2 = axs[2].imshow(u2.reshape(n, n), origin='lower', extent=[0,1,0,1],
                    cmap='viridis', vmin=umin, vmax=umax)
axs[2].set_title('u2 (from f2)')
axs[2].set_xlabel('x')
axs[2].set_ylabel('y')
plt.colorbar(im2, ax=axs[2])

plt.tight_layout()
plt.show()
```

결과는 다음과 같다.



또한 포아송 방정식의 선형성을 확인하기 위해 u-u1-u2가 0에 근접한다는 것을 보이기 위해 다음과 같은 코드를 작성했다.

```
# u - u1 - u2 계산
error = u - u1 - u2
error_2D = error.reshape(n, n)

# 오차의 최대값 출력
print("max error = ", np.max(np.abs(error)))

# 오차 시각화
plt.figure(figsize=(6, 5))
plt.imshow(error_2D, origin='lower', extent=[0, 1, 0, 1], cmap='seismic')
plt.colorbar(label='Error magnitude')
plt.title('u - u1 - u2')
plt.xlabel('x')
plt.ylabel('y')
plt.tight_layout()
plt.show()
```

결과는 다음과 같이 오차의 최대값이 매우 작았다.

max error = 5.284844259150368e-06