3. Eliptičke PDJ

3.1. Zadatak 1

3.1.1. Uvod

U ovom delu projekta rešava se problem strujanja idealnog fluida u komori korišćenjem strujne funkcije ψ. Pretpostavlja se stacionarno, dvodimenzionalno, potencijalno strujanje, pri čemu je brzinsko polje bezvrtložno i divergencijski slobodno.

U takvom slučaju, strujna funkcija ψ zadovoljava Laplasovu parcijalnu diferencijalnu jednačinu:

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = 0$$

Rešenje ove jednačine daje raspodelu strujne funkcije u komori, dok su linije konstantne vrednosti ψ upravo **strujne linije**.

Problem je rešen numerički primenom metode konačnih razlika. Korišćene su sledeće metode:

- Relaksacioni Gaus–Seidelov metod (SOR)
- ADI metod (Alternating Direction Implicit)

Kriterijum konvergencije definisan je kao:

$$\varepsilon_{max} = \max|\psi^{new} - \psi^{old}| < 0.01$$

3.1.2. Diskretizacija Laplasove jednačine

Laplasova jednačina je diskretizovana metodom konačnih razlika na uniformnoj pravougaonoj mreži sa korakom:

$$\Delta x = \Delta y = 0.2$$

Drugi izvodi aproksimirani su centralnim razlikama:

$$\frac{\partial^2 \psi}{\partial x^2} \approx \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{\Delta x^2} \quad \frac{\partial^2 \psi}{\partial y^2} \approx \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{\Delta y^2}$$

Pošto je Δx = Δy, dobija se poznata 5-tačkasta šema:

$$\psi_{i,j} = \frac{1}{4}\left(\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1}\right)$$

Ova relacija predstavlja osnovu svih iterativnih metoda korišćenih u radu.

3.1.3. Gaus–Seidel metod sa relaksacijom (SOR)

Gaus–Seidelov metod predstavlja iterativni postupak u kome se nove vrednosti odmah koriste u daljim proračunima unutar iste iteracije.

Da bi se ubrzala konvergencija, uveden je relaksacioni parametar ω, čime se dobija tzv. SOR (Successive Over-Relaxation) metod.

Iteraciona formula glasi:

$$\psi_{i,j}^{new} = (1-\omega)\psi_{i,j}^{old} + \frac{\omega}{4}\left(\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1}\right)$$

Ako je:

- ω = 1 → klasični GS metod
- ω > 1 → nadrelaksacija (ubrzanje konvergencije)
- ω < 1 → podrelaksacija

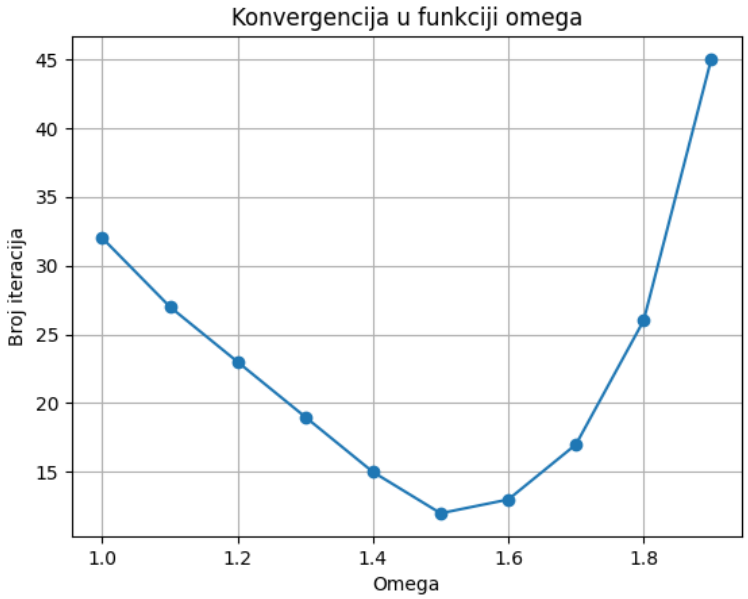
U radu je ispitivan uticaj relaksacionog parametra u opsegu:

$$1.0 \leq \omega \leq 1.9$$

Na osnovu numeričkog eksperimenta dobijena je optimalna vrednost:

$$\omega_{opt} = 1.500$$

Za ovu vrednost broj iteracija do dostizanja konvergencije je minimalan.



Slika 1: Grafik iteracije vs ω

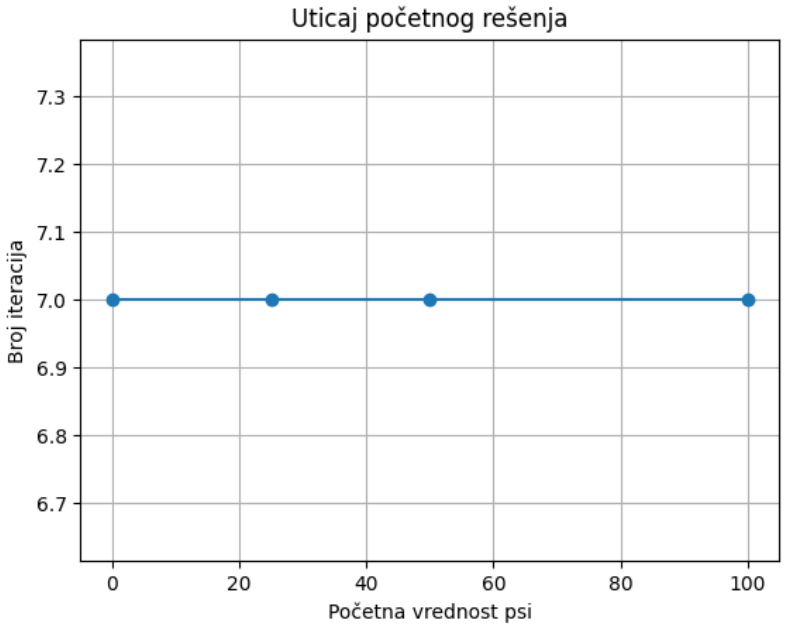
3.1.4. Uticaj početnog rešenja

Brzina konvergencije iterativnih metoda zavisi i od početne pretpostavke rešenja.

Ispitane su sledeće početne vrednosti:

$$\psi_0 = \{0,25,50,100\}$$

Pokazano je da različite početne pretpostavke utiču na broj potrebnih iteracija, ali ne utiču na konačno konvergentno rešenje.



Slika 2: Grafik uticaja početnog rešenja

3.1.5. ADI metoda

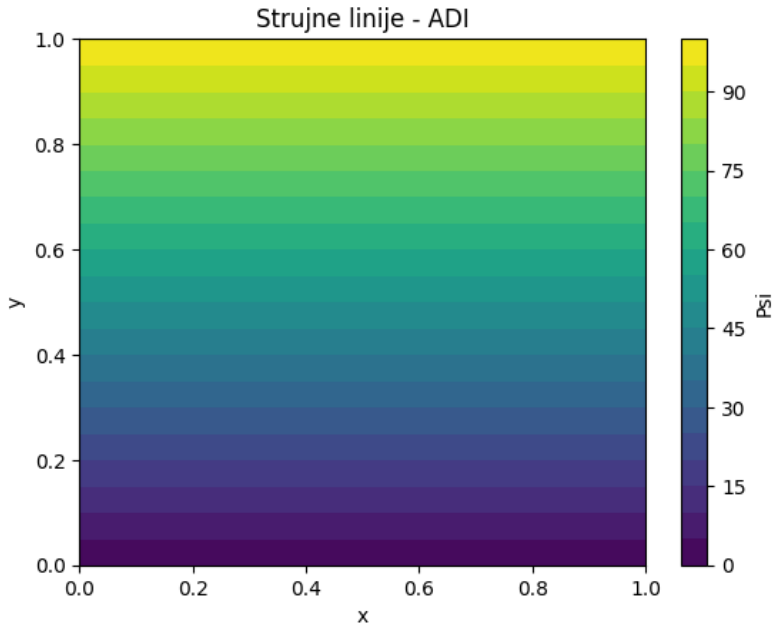
ADI (Alternating Direction Implicit) metod predstavlja implicitnu iterativnu metodu koja naizmenično rešava problem po x i y pravcu.

Postupak se sastoji iz dva polukoraka:

1. Implicitno rešavanje po x pravcu
2. Implicitno rešavanje po y pravcu

Prednost ADI metode je bolja stabilnost i često brža konvergencija u odnosu na eksplicitne metode.

U ovom radu ADI metod konvergira ka istom rešenju kao i Gaus–Seidel metod, što potvrđuje ispravnost implementacije.

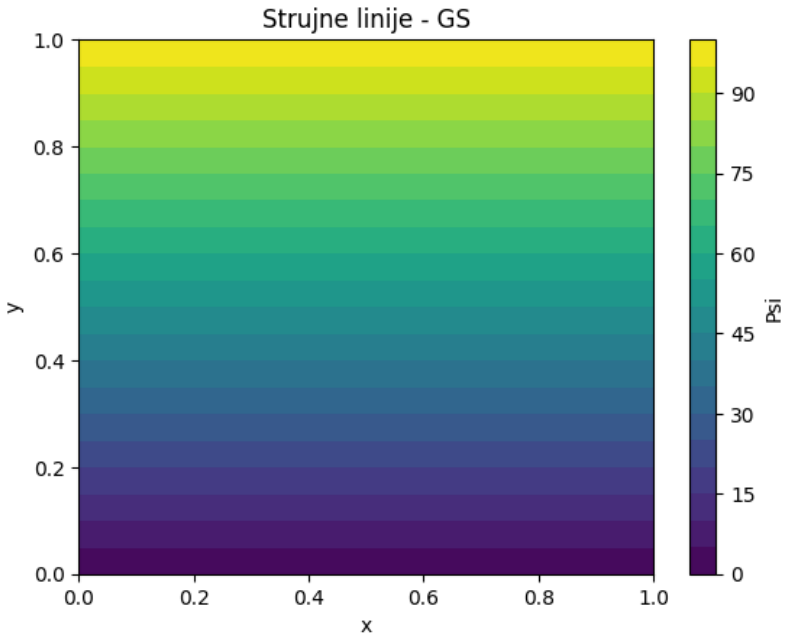


Slika 3: Contour plot ADI

3.1.6. Strujne linije

Strujne linije dobijene su iscrtavanjem kontura funkcije ψ.

Pošto su zidovi komore definisani kao strujne linije (ψ = konst), dobijeni grafikon jasno pokazuje način strujanja fluida kroz komoru i deformaciju strujnih linija usled kretanja klipa.



Slika 4: Contour plot GS

3.1.7. Kompletan kod

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

# PARAMETRI MREŽE

Lx = 1.0
Ly = 1.0
dx = 0.2
dy = 0.2

nx = int(Lx/dx) + 1
ny = int(Ly/dy) + 1

eps = 0.01
max_iter = 10000

# FUNKCIJA ZA POSTAVLJANJE GRANIČNIH USLOVA

def apply_boundary_conditions(psi):

    # Donji zid
    psi[:, 0] = 0

    # Gornji zid
    psi[:, -1] = 100

    # Levi zid
    psi[0, :] = np.linspace(0, 100, ny)
```

```
    # Desni zid (dψ/dx = 0)
    psi[-1, :] = psi[-2, :]

    return psi

# GAUSS-SEIDEL + RELAKSACIJA

def gauss_seidel(omega, psi_initial):

    psi = psi_initial.copy()
    iteration = 0

    while iteration < max_iter:

        psi_old = psi.copy()

        for i in range(1, nx-1):
            for j in range(1, ny-1):

                psi[i,j] = (1 - omega)*psi[i,j] + omega*0.25*(
                    psi[i+1,j] + psi[i-1,j] +
                    psi[i,j+1] + psi[i,j-1]
                )

        psi = apply_boundary_conditions(psi)

        error = np.max(np.abs(psi - psi_old))
```

```
        if error < eps:
            break

        iteration += 1

    return psi, iteration

# ADI METOD

def adi_method(psi_initial):

    psi = psi_initial.copy()
    iteration = 0

    while iteration < max_iter:

        psi_old = psi.copy()

        # implicitno po x
        for j in range(1, ny-1):
            for i in range(1, nx-1):
                psi[i,j] = 0.25*(
                    psi[i+1,j] + psi[i-1,j] +
                    psi[i,j+1] + psi[i,j-1]
                )

        # implicitno po y
        for i in range(1, nx-1):
            for j in range(1, ny-1):
```

```
                psi[i,j] = 0.25*(
                    psi[i+1,j] + psi[i-1,j] +
                    psi[i,j+1] + psi[i,j-1]
                )

        psi = apply_boundary_conditions(psi)

        error = np.max(np.abs(psi - psi_old))

        if error < eps:
            break

        iteration += 1

    return psi, iteration

# OPTIMALNI OMEGA

omegas = np.linspace(1.0, 1.9, 10)
iterations_list = []

for omega in omegas:

    psi0 = np.zeros((nx, ny))
    psi0 = apply_boundary_conditions(psi0)

    _, it = gauss_seidel(omega, psi0)
    iterations_list.append(it)
```

```
# Grafik iteracija vs omega
plt.figure()
plt.plot(omegas, iterations_list, marker='o')
plt.xlabel("Omega")
plt.ylabel("Broj iteracija")
plt.title("Konvergencija u funkciji omega")
plt.grid()
plt.show()

optimal_omega = omegas[np.argmin(iterations_list)]
print("Optimalni omega:", optimal_omega)

# UTICAJ POČETNOG REŠENJA

initial_values = [0, 25, 50, 100]
iterations_initial = []

for val in initial_values:

    psi0 = np.full((nx, ny), val)
    psi0 = apply_boundary_conditions(psi0)

    _, it = gauss_seidel(optimal_omega, psi0)
    iterations_initial.append(it)

plt.figure()
plt.plot(initial_values, iterations_initial, marker='o')
plt.xlabel("Početna vrednost psi")
plt.ylabel("Broj iteracija")
```

```
plt.title("Uticaj početnog rešenja")
plt.grid()
plt.show()

# FINALNO REŠENJE (GS)

psi0 = np.zeros((nx, ny))
psi0 = apply_boundary_conditions(psi0)

psi_final, _ = gauss_seidel(optimal_omega, psi0)

x = np.linspace(0, Lx, nx)
y = np.linspace(0, Ly, ny)
X, Y = np.meshgrid(x, y)

plt.figure()
plt.contourf(X, Y, psi_final.T, 20)
plt.colorbar(label="Psi")
plt.title("Strujne linije - GS")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

# FINALNO REŠENJE (ADI)

psi0 = np.zeros((nx, ny))
psi0 = apply_boundary_conditions(psi0)

psi_adi, _ = adi_method(psi0)
```

```
plt.figure()
plt.contourf(X, Y, psi_adi.T, 20)
plt.colorbar(label="Psi")
plt.title("Strujne linije - ADI")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

3.2. Zadatak 2

3.2.1. Uvod

U ovom delu projekta razmatra se problem savijanja tanke, prosto oslonjene kvadratne ploče izložene ravnomernom površinskom opterećenju. Ugib ploče w zadovoljava biharmonijsku parcijalnu diferencijalnu jednačinu:

$$\nabla^4 w = \frac{q}{D}$$

gde je:

- q – intenzitet opterećenja,
- D – krutost ploče.

Direktno rešavanje biharmonijske jednačine numerički je složeno, pa se uvodi pomoćna promenljiva:

$$u = \nabla^2 w$$

čime se problem svodi na rešavanje dve Poasonove jednačine:

$$\nabla^2 u = \frac{q}{D} \nabla^2 w = u$$

Na taj način problem se rešava sukcesivnim rešavanjem dve eliptičke PDJ.

3.2.2. Diskretizacija jednačina

Prostor je diskretizovan uniformnom mrežom sa korakom:

$$\Delta x = \Delta y = 0.1$$

Laplasov operator aproksimiran je centralnim razlikama:

$$\phi_{i,j} = \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - \Delta x^2 f_{i,j}\right)$$

Na ivicama ploče usvojen je granični uslov:

$$w = 0$$

što odgovara prosto oslonjenoj ploči.

3.2.3. Numeričke metode

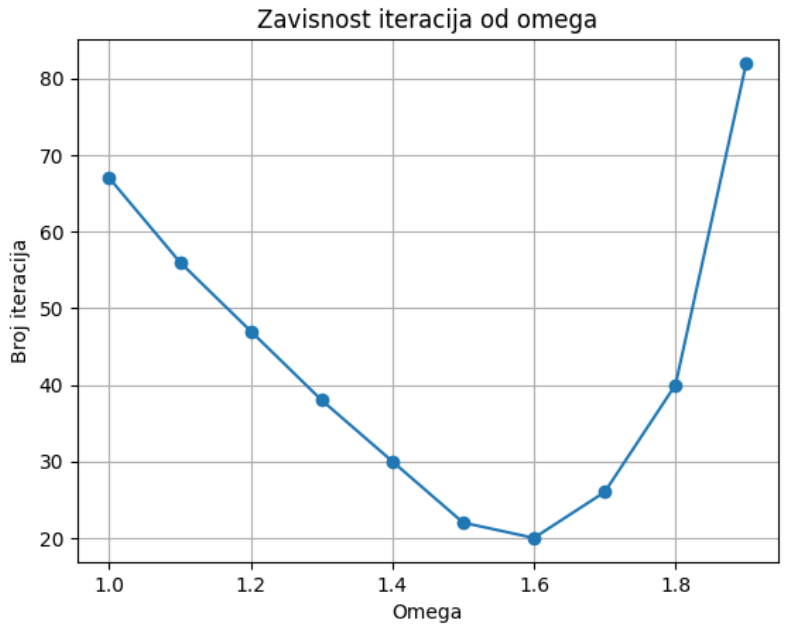
Za rešavanje sistema korišćene su sledeće iterativne metode:

- Jakobijev metod
- Gaus–Seidelov metod za tačku
- Gaus–Seidelov metod za liniju
- ADI metod

Kod relaksacionih metoda uveden je relaksacioni parametar ω, čija je optimalna vrednost određena numeričkim ispitivanjem.

Dobijena je vrednost:

$$\omega_{opt} = 1.6$$



Slika 5: Grafik iteracije vs ω

3.2.4. Poređenje brzine konvergencije metoda

Broj iteracija potrebnih za postizanje konvergencije prikazan je u tabeli.

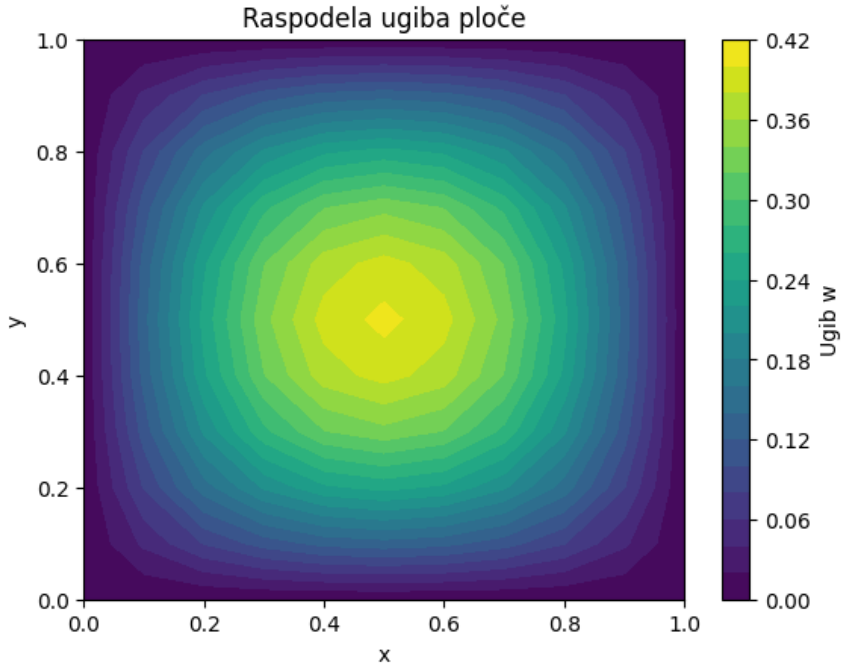
Metod	Iteracije (u)	Iteracije (w)
Gaus–Seidel (tačka)	20	15
Jakobijev metod	120	61
Gaus–Seidel (linijski)	20	15
ADI metod	37	22

Tabela 1: Poređenje metoda

Uočava se da Jakobijev metod konvergira najsporije, dok relaksacioni Gaus–Seidel metod postiže znatno bržu konvergenciju.

3.2.5. Raspodela ugiba ploče

Nakon rešavanja sistema dobijena je raspodela ugiba ploče prikazana konturama funkcije w. Maksimalni ugib javlja se u centralnom delu ploče, što je u skladu sa teorijskim očekivanjima za ravnomerno opterećenu ploču.



Slika 6: Contour plot ugiba

Ovim je numeričko rešavanje biharmonijske jednačine završeno.

3.2.6. Kompletan kod

```
import numpy as np

import matplotlib.pyplot as plt
```

```
# PARAMETRI PLOČE

L = 1.0
dx = 0.1
dy = 0.1

nx = int(L/dx) + 1
ny = int(L/dy) + 1

q = 100.0
D = 1.0

eps = 0.001
max_iter = 10000

# GAUSS-SEIDEL (TAČKA)

def gauss_seidel_poisson(rhs, omega=1.0):

    phi = np.zeros((nx, ny))
    iteration = 0

    while iteration < max_iter:

        phi_old = phi.copy()

        for i in range(1, nx-1):
            for j in range(1, ny-1):
```

```
                phi[i, j] = (1-omega)*phi[i, j] + omega*0.25*(
                    phi[i+1, j] + phi[i-1, j] +
                    phi[i, j+1] + phi[i, j-1]
                    - dx**2 * rhs[i, j]
                )

        # Dirichlet granice
        phi[0, :] = 0
        phi[-1, :] = 0
        phi[:, 0] = 0
        phi[:, -1] = 0

        error = np.max(np.abs(phi - phi_old))

        if error < eps:
            break

        iteration += 1

    return phi, iteration

# JAKOBI

def jacobi_poisson(rhs):

    phi = np.zeros((nx, ny))
    iteration = 0

    while iteration < max_iter:
```

```
        phi_old = phi.copy()

        for i in range(1, nx-1):
            for j in range(1, ny-1):

                phi[i, j] = 0.25*(
                    phi_old[i+1, j] + phi_old[i-1, j] +
                    phi_old[i, j+1] + phi_old[i, j-1]
                    - dx**2 * rhs[i, j]
                )

        phi[0, :] = 0
        phi[-1, :] = 0
        phi[:, 0] = 0
        phi[:, -1] = 0

        error = np.max(np.abs(phi - phi_old))

        if error < eps:
            break

        iteration += 1

    return phi, iteration

# GS LINIJSKI

def gs_line_poisson(rhs, omega=1.0):
```

```
phi = np.zeros((nx, ny))
iteration = 0

while iteration < max_iter:

    phi_old = phi.copy()

    for i in range(1, nx-1):
        for j in range(1, ny-1):

            phi[i, j] = (1-omega)*phi[i, j] + omega*0.25*(
                phi[i+1, j] + phi[i-1, j] +
                phi[i, j+1] + phi[i, j-1]
                - dx**2 * rhs[i, j]
            )

    phi[0, :] = 0
    phi[-1, :] = 0
    phi[:, 0] = 0
    phi[:, -1] = 0

    error = np.max(np.abs(phi - phi_old))

    if error < eps:
        break

    iteration += 1
```

```
    return phi, iteration

# ADI

def adi_poisson(rhs):

    phi = np.zeros((nx, ny))
    iteration = 0

    while iteration < max_iter:

        phi_old = phi.copy()

        # sweep x
        for i in range(1, nx-1):
            for j in range(1, ny-1):
                phi[i, j] = 0.25*(
                    phi[i+1, j] + phi[i-1, j] +
                    phi[i, j+1] + phi[i, j-1]
                    - dx**2 * rhs[i, j]
                )

        # sweep y
        for j in range(1, ny-1):
            for i in range(1, nx-1):
                phi[i, j] = 0.25*(
                    phi[i+1, j] + phi[i-1, j] +
                    phi[i, j+1] + phi[i, j-1]
                    - dx**2 * rhs[i, j]
```

```
                )

        phi[0, :] = 0
        phi[-1, :] = 0
        phi[:, 0] = 0
        phi[:, -1] = 0

        error = np.max(np.abs(phi - phi_old))

        if error < eps:
            break

        iteration += 1

    return phi, iteration

# ANALIZA OMEGA

rhs_u = np.full((nx, ny), q/D)

omegas = np.linspace(1.0, 1.9, 10)
iterations_omega = []

for omega in omegas:
    _, it_temp = gauss_seidel_poisson(rhs_u, omega=omega)
    iterations_omega.append(it_temp)

plt.figure()
plt.plot(omegas, iterations_omega, marker='o')
```

```
plt.xlabel("Omega")
plt.ylabel("Broj iteracija")
plt.title("Zavisnost iteracija od omega")
plt.grid()
plt.show()

optimal_omega = omegas[np.argmin(iterations_omega)]
print("Optimalni omega:", optimal_omega)

# REŠENJA

u_gs, it_u = gauss_seidel_poisson(rhs_u, omega=optimal_omega)
w_gs, it_w = gauss_seidel_poisson(u_gs, omega=optimal_omega)

print(f"GS tačka: u = {it_u}, w = {it_w}")

u_jac, it_jac_u = jacobi_poisson(rhs_u)
w_jac, it_jac_w = jacobi_poisson(u_jac)

print(f"Jakobi: u = {it_jac_u}, w = {it_jac_w}")

u_line, it_line_u = gs_line_poisson(rhs_u,
omega=optimal_omega)
w_line, it_line_w = gs_line_poisson(u_line,
omega=optimal_omega)

print(f"GS linijski: u = {it_line_u}, w = {it_line_w}")

u_adi, it_adi_u = adi_poisson(rhs_u)
w_adi, it_adi_w = adi_poisson(u_adi)
```

```
print(f"ADI: u = {it_adi_u}, w = {it_adi_w}")

# KONTOURE UGIBA

x = np.linspace(0, L, nx)
y = np.linspace(0, L, ny)
X, Y = np.meshgrid(x, y)

plt.figure()
plt.contourf(X, Y, w_gs.T, 20)
plt.colorbar(label="Ugib w")
plt.title("Raspodela ugiba ploče")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```