Numerical analysis calculation work

Ivan Zhytkevych FI-91

Методи обчислень. Розрахунково-графічна робота.

Варіант 4 *Змоделювати процес розповсюдження газоподібної забруднюючої домішки в атмосфері за умови наявності ефекту самоочищення середовища.*
Самоочищення виникає при досягненні концентрацією u деякого критичного значення.

Рівняння еволюції концентрації домішки має вигляд:

$$\frac{\partial u}{\partial t} = k_1 \frac{\partial^2 u}{\partial x^2} + k_2 \frac{\partial^2 u}{\partial y^2} + \mathcal{X} + \varphi$$

$$\mathcal{X} = \begin{cases} 0 & u < u_{cr} \\ -du & u \geq u_{cr} \end{cases}$$

$$u(0) = 0$$

$$u\,|_\Gamma = 0$$

Прийняти, що джерела забруднення розміщені в двох несусідніх центральних точках області, і діють з інтенсивністю $\varphi = 5$мкг/хв. Розміри області завдати самостійно (але розбиття області повинно включати не менше 10 вузлів за кожним напрямком). Знайти поле концентрації домішки на протязі часу, за який концентрація домішки встигне досягти свого критичного значення хоча б в деяких точках області.

$$t \in [0, \infty]$$

Нехай маємо границі по осі x: $[a_x, b_x]$, та по осі y: $[a_y, b_y]$. Тоді:

$$u\,|_\Gamma = 0 \;\Rightarrow\; u(a, b) = 0 \quad \forall a \in \{a_x, a_y\}, \forall b \in \{b_x, b_y\}$$

Дискретизація

Рівняння має параболічний тип. Будемо використовувати неявну схему Кранка-Ніколсона:

- розбиваємо відрізки $[a_x, b_x]$ та $[a_y, b_y]$ на $n + 1$ вузлів
- i позначатимемо номер вузла по $[a_x, b_x]$, а j по $[a_y, b_y]$
- відстань між сусідніми вузлами Δx та Δy

- дискретизуємо час t: $t_h \in [0, m]$ та індекс $h \in \mathbb{N}_0, h = \overline{0, l}$. крок Δt
- джерело забруднення: нехай маємо дві пари (s_1, d_1) та (s_2, d_2), такі що
  - $s_2 - s_1 > 1 \ \wedge \ d_2 - d_1 > 1$ - умова несусідства тоді

$$\varphi = \begin{cases} 0 & (i,j) \neq (s_1, d_1) \ \wedge \ (i,j) \neq (s_2, d_2) \\ 5 & (i,j) = (s_1, d_1) \ \vee \ (i,j) = (s_2, d_2) \end{cases}$$

Беремо скінченно-різницеві апроксимації:

$$\frac{\partial u}{\partial t} = \frac{u_{i,j}^{h+1} - u_{i,j}^{h}}{\Delta t}$$

$$\frac{\partial^2 u}{\partial x^2} = \lambda \frac{u_{i+1,j}^{h+1} - 2u_{i,j}^{h+1} + u_{i-1,j}^{h+1}}{\Delta x^2} + (1-\lambda)\frac{u_{i+1,j}^{h} - 2u_{i,j}^{h} + u_{i-1,j}^{h}}{\Delta x^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \lambda \frac{u_{i,j+1}^{h+1} - 2u_{i,j}^{h+1} + u_{i,j-1}^{h+1}}{\Delta x^2} + (1-\lambda)\frac{u_{i,j+1}^{h} - 2u_{i,j}^{h} + u_{i,j-1}^{h}}{\Delta y^2}$$

$$\frac{\partial u}{\partial x} = \lambda \frac{u_{i+1,j}^{h+1} - u_{i,j}^{h+1}}{\Delta x} + (1-\lambda)\frac{u_{i+1,j}^{h} - u_{i,j}^{h}}{\Delta x}$$
$$\frac{\partial u}{\partial y} = \lambda \frac{u_{i,j+1}^{h+1} - u_{i,j}^{h+1}}{\Delta y} + (1-\lambda)\frac{u_{i,j+1}^{h} - u_{i,j}^{h}}{\Delta y}$$
$$\lambda \in (0,1)$$

Різницева задача

$$\frac{u_{i,j}^{h+1} - u_{i,j}^{h}}{\Delta t} = k_1 \left(\lambda \frac{u_{i+1,j}^{h+1} - 2u_{i,j}^{h+1} + u_{i-1,j}^{h+1}}{\Delta x^2} + (1-\lambda)\frac{u_{i+1,j}^{h} - 2u_{i,j}^{h} + u_{i-1,j}^{h}}{\Delta x^2} \right) +$$
$$+ k_2 \left(\lambda \frac{u_{i,j+1}^{h+1} - 2u_{i,j}^{h+1} + u_{i,j-1}^{h+1}}{\Delta y^2} + (1-\lambda)\frac{u_{i,j+1}^{h} - 2u_{i,j}^{h} + u_{i,j-1}^{h}}{\Delta y^2} \right) + \mathcal{X} + \varphi$$

$$\mathcal{X} = \begin{cases} 0 & u < \\ -\left(\lambda\left(u_{i+1,j}^{h+1} + u_{i,j+1}^{h+1} - 2u_{i,j}^{h+1}\right) + (1-\lambda)\left(u_{i+1,j}^{h} + u_{i,j+1}^{h} - 2u_{i,j}^{h}\right) + u_{i,j}^{h+1} - u_{i,j}^{h}\right) & u \geq \end{cases}$$

$\exists (s_1, d_1)$ and $\exists (s_2, d_2)$: $\quad s_2 - s_1 > 1 \ \wedge \ d_2 - d_1 > 1$

$$\varphi = \begin{cases} 0 & (i,j) \neq (s_1, d_1) \ \wedge \ (i,j) \neq (s_2, d_2) \\ 5 & (i,j) = (s_1, d_1) \ \vee \ (i,j) = (s_2, d_2) \end{cases}$$

З початковими:

$$\forall i, j \quad u_{i,j}^{0} = 0$$

З граничними:

$$\forall h: \ \forall i : u^h_{i,0} = u^h_{i,n} = 0$$
$$\forall h: \ \forall j : u^h_{0,j} = u^h_{n,j} = 0$$

Перенесемо всі значення u у момент $h+1$ у ліву частину рівняння:

$$\frac{1}{\Delta t}u^{h+1}_{i,j} - k_1\frac{\lambda}{\Delta x^2}\left(u^{h+1}_{i+1,j} - 2u^{h+1}_{i,j} + u^{h+1}_{i-1,j}\right) - k_2\frac{\lambda}{\Delta y^2}\left(u^{h+1}_{i,j+1} - 2u^{h+1}_{i,j} + u^{h+1}_{i,j-1}\right) =$$

$$= \frac{1}{\Delta t}u^h_{i,j} + k_1\frac{(1-\lambda)}{\Delta x^2}\left(u^h_{i+1,j} - 2u^h_{i,j} + u^h_{i-1,j}\right) + k_2\frac{(1-\lambda)}{\Delta y^2}\left(u^h_{i,j+1} - 2u^h_{i,j} + u^h_{i,j-1}\right) + \mathcal{X} + \varphi$$

Але маємо два випадки у залежності від u:

якщо $u < u_{cr}$, то

$$\frac{1}{\Delta t}u^{h+1}_{i,j} - k_1\frac{\lambda}{\Delta x^2}\left(u^{h+1}_{i+1,j} - 2u^{h+1}_{i,j} + u^{h+1}_{i-1,j}\right) - k_2\frac{\lambda}{\Delta y^2}\left(u^{h+1}_{i,j+1} - 2u^{h+1}_{i,j} + u^{h+1}_{i,j-1}\right) =$$

$$= \frac{1}{\Delta t}u^h_{i,j} + k_1\frac{(1-\lambda)}{\Delta x^2}\left(u^h_{i+1,j} - 2u^h_{i,j} + u^h_{i-1,j}\right) + k_2\frac{(1-\lambda)}{\Delta y^2}\left(u^h_{i,j+1} - 2u^h_{i,j} + u^h_{i,j-1}\right) + \varphi$$

$$\left(-\frac{k_1\lambda}{\Delta x^2}\right)u^{h+1}_{i+1,j} + \left(-\frac{k_2\lambda}{\Delta y^2}\right)u^{h+1}_{i,j+1} + \left(\frac{1}{\Delta t} + \frac{2k_1\lambda}{\Delta x^2} + \frac{2k_2\lambda}{\Delta y^2}\right)u^{h+1}_{i,j} +$$
$$+ \left(-\frac{k_1\lambda}{\Delta x^2}\right)u^{h+1}_{i-1,j} + \left(-\frac{k_2\lambda}{\Delta y^2}\right)u^{h+1}_{i,j-1} =$$

$$= \left(\frac{k_1(1-\lambda)}{\Delta x^2}\right)u^h_{i+1,j} + \left(\frac{k_2(1-\lambda)}{\Delta y^2}\right)u^h_{i,j+1} + \left(\frac{1}{\Delta t} - \frac{2k_1(1-\lambda)}{\Delta x^2} - \frac{2k_2(1-\lambda)}{\Delta y^2}\right)u^h_{i,j} +$$
$$+ \left(\frac{k_1(1-\lambda)}{\Delta x^2}\right)u^h_{i-1,j} + \left(\frac{k_2(1-\lambda)}{\Delta y^2}\right)u^h_{i,j-1} + \varphi$$

а якщо $u \geq u_{cr}$:

$$\frac{1}{\Delta t}u^{h+1}_{i,j} - k_1\frac{\lambda}{\Delta x^2}\left(u^{h+1}_{i+1,j} - 2u^{h+1}_{i,j} + u^{h+1}_{i-1,j}\right) - k_2\frac{\lambda}{\Delta y^2}\left(u^{h+1}_{i,j+1} - 2u^{h+1}_{i,j} + u^{h+1}_{i,j-1}\right) =$$

$$= \frac{1}{\Delta t}u^h_{i,j} + k_1\frac{(1-\lambda)}{\Delta x^2}\left(u^h_{i+1,j} - 2u^h_{i,j} + u^h_{i-1,j}\right) + k_2\frac{(1-\lambda)}{\Delta y^2}\left(u^h_{i,j+1} - 2u^h_{i,j} + u^h_{i,j-1}\right) -$$

$$-\left(\lambda\left(u_{i+1,j}^{h+1}+u_{i,j+1}^{h+1}-2u_{i,j}^{h+1}\right)+(1-\lambda)\left(u_{i+1,j}^{h}+u_{i,j+1}^{h}-2u_{i,j}^{h}\right)+u_{i,j}^{h+1}-u_{i,j}^{h}\right)$$

$$\left(-\frac{k_1\lambda}{\Delta x^2}+\lambda\right)u_{i+1,j}^{h+1}+\left(-\frac{k_2\lambda}{\Delta y^2}+\lambda\right)u_{i,j+1}^{h+1}+\left(\frac{1}{\Delta t}+\frac{2k_1\lambda}{\Delta x^2}+\frac{2k_2\lambda}{\Delta y^2}-2\lambda+1\right)u_{i,j}^{h+1}+$$

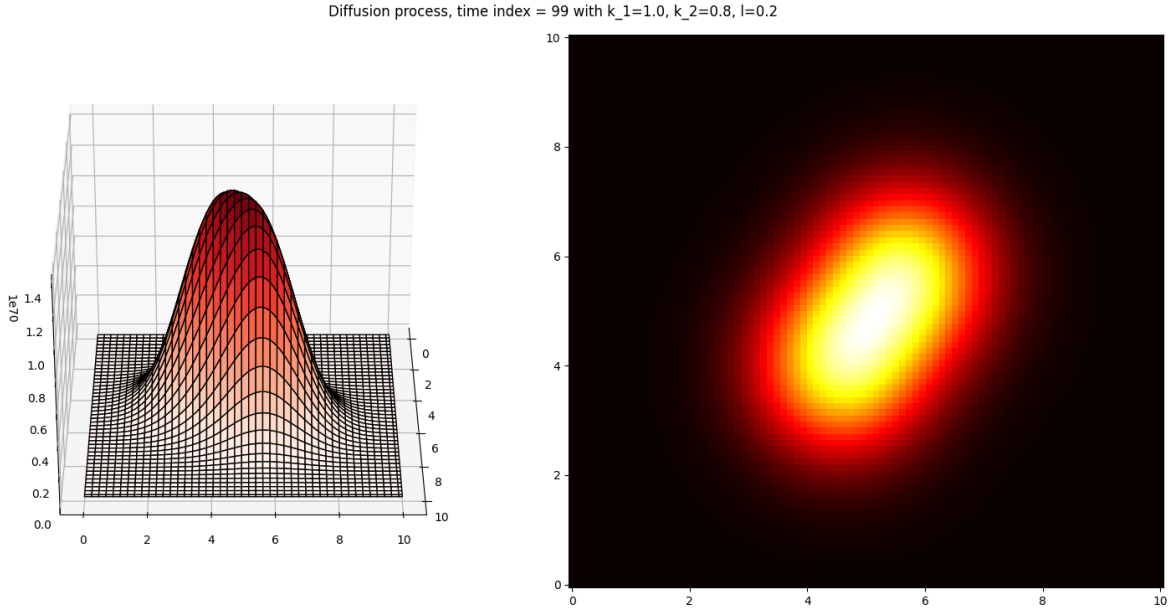
$$+\left(-\frac{k_1\lambda}{\Delta x^2}\right)u_{i-1,j}^{h+1}+\left(-\frac{k_2\lambda}{\Delta y^2}\right)u_{i,j-1}^{h+1}=$$

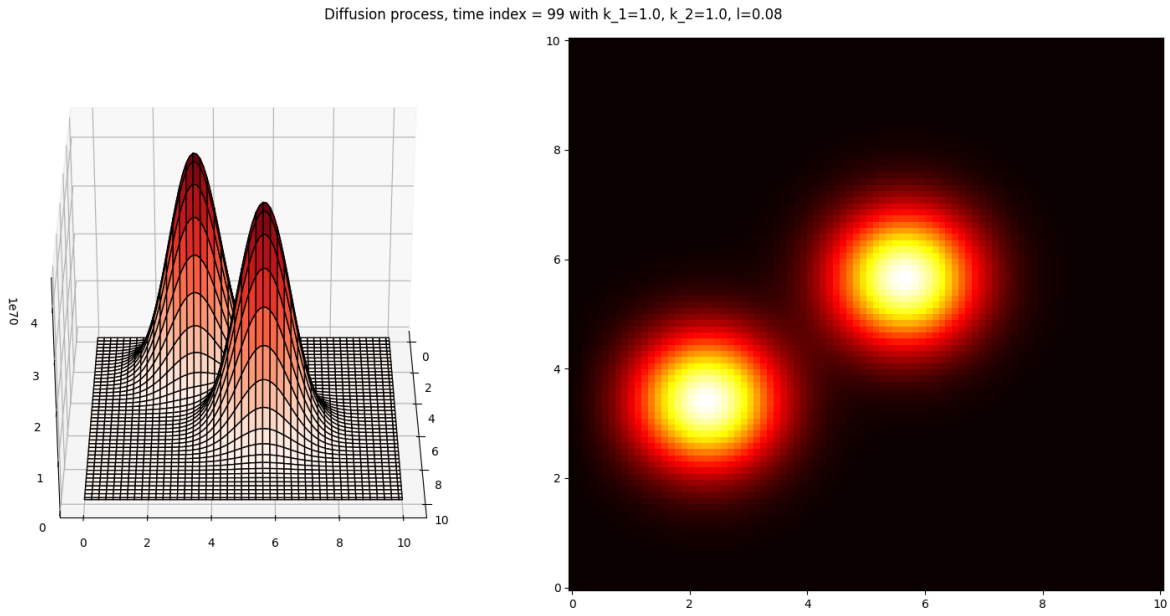
$$=\left(\frac{k_1(1-\lambda)}{\Delta x^2}-1+\lambda\right)u_{i+1,j}^{h}+\left(\frac{k_2(1-\lambda)}{\Delta y^2}-1+\lambda\right)u_{i,j+1}^{h}+$$

$$+\left(\frac{1}{\Delta t}-\frac{2k_1(1-\lambda)}{\Delta x^2}-\frac{2k_2(1-\lambda)}{\Delta y^2}+2-2\lambda+1\right)u_{i,j}^{h}+$$

$$+\left(\frac{k_1(1-\lambda)}{\Delta x^2}\right)u_{i-1,j}^{h}+\left(\frac{k_2(1-\lambda)}{\Delta y^2}\right)u_{i,j-1}^{h}+\varphi$$

Результати





Анімації процесу можна знайти по посиланню

Code

```
import sys
import time
import itertools

import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt
from matplotlib import animation
```

```
from mpl_toolkits.mplot3d import Axes3D


class DiffusionDiffEquationSolver:
    emitters = []
    area = {}
    history = {}
    current_time = 0
    lattice = None

    def __init__(self, k_1, k_2, l,
                 n, x, y, t_step,
                 u_crit,
                 emitters, phi,
                 initial):
        self.k_1 = k_1
        self.k_2 = k_2
        self.l = l
        self.n = n

        self.area['x'] = x
        self.area['y'] = y
        self.nodesx = np.linspace(self.area['x'][0], self.area['x'][1], n)
        self.nodesy = np.linspace(self.area['y'][0], self.area['y'][1], n)

        self.nodes = np.array(list(itertools.product([i for i in range(1, n-1)], repeat=2)))

        self.x_step = self.nodesx[1] - self.nodesx[0]
        self.y_step = self.nodesy[1] - self.nodesy[0]
        self.t_step = t_step
        if np.power(self.x_step, 2) * 0.4 < self.t_step:
            print('The paramethers lead to unstable model')
            self.t_step = np.power(self.x_step , 2)

        self.u_crit = u_crit
        self.phi = phi
        self.emitters = emitters
        # unitialize lattice
        self.lattice = np.zeros((n, n))
        for i in range(self.lattice.shape[0]):
            for j in range(self.lattice.shape[1]):
                self.lattice[i, j] = initial(i, j)

    def _build_discrete_eq_coefs_next(self, i, j):
        coefs = np.array([
            [- self.k_1 * self.l / np.power(self.x_step, 2)],
            [- self.k_2 * self.l / np.power(self.y_step, 2)],
            [1/self.t_step +
             2 * self.k_1 * (1-self.l)/np.power(self.x_step, 2) -
             2 * self.k_2 * (1-self.l)/np.power(self.y_step, 2)],
```

```
            [- self.k_1 * self.l / np.power(self.x_step, 2)],
            [- self.k_2 * self.l / np.power(self.y_step, 2)],
        ])

        clearance_strength = 1
        if self.lattice[i, j] >= self.u_crit:
            coefs[0, 0] += self.l * clearance_strength
            coefs[1, 0] += self.l * clearance_strength
            coefs[2, 0] += (1 - 2 * self.l) * clearance_strength
        return coefs

    def _build_discrete_coefs_bias(self, i, j):
        coefs = np.array([
            [self.k_1 * (1 - self.l) / np.power(self.x_step, 2)],
            [self.k_2 * (1 - self.l) / np.power(self.y_step, 2)],
            [1/self.t_step -
             2 * self.k_1 * self.l / np.power(self.x_step, 2) +
             2 * self.k_2 * self.l / np.power(self.y_step, 2)],
            [self.k_1 * (1 - self.l) / np.power(self.x_step, 2)],
            [self.k_2 * (1 - self.l) / np.power(self.y_step, 2)],
        ])
        clearance_strength = 1
        clearance_addition = 0
        if self.lattice[i, j] >= self.u_crit:
            coefs[0, 0] += (- 1 + self.l) * clearance_strength + clearance_addition
            coefs[1, 0] += (- 1 + self.l) * clearance_strength + clearance_addition
            coefs[2, 0] += (3 - 2 * self.l) * clearance_strength + clearance_addition
        return coefs

    def _emition_at(self, i, j):
        if (i, j) in self.emitters:
            return self.phi
        return 0

    def _compute_bias_at(self, i, j):
        coefs = self._build_discrete_coefs_bias(i, j)
        concentration = np.array([
            [self.lattice[i+1, j]],
            [self.lattice[i, j+1]],
            [self.lattice[i, j]],
            [self.lattice[i-1, j]],
            [self.lattice[i, j-1]],
        ])
        return np.sum(coefs * concentration) + self._emition_at(i, j)

    def _build_next_system(self):
        n = np.power(self.n - 2, 2)
        next_lattice_coefs = np.zeros((n, n))

        def commit_to_next_lattice(eqn, i, j, value):
```

```
            variable_n = (self.n - 2) * (i - 1) + (j - 1)
            if i == 0 or j == 0 or i == self.n-1 or j == self.n-1:
                return
            next_lattice_coefs[eqn, variable_n] = value

        bias = np.zeros((n, 1))
        for eqn, node_idx in enumerate(self.nodes):
            b = self._compute_bias_at(node_idx[0], node_idx[1])
            row_idx = (self.n - 2) * (node_idx[0] - 1) + (node_idx[1] - 1)
            bias[row_idx, 0] = b
            left_coefs = self._build_discrete_eq_coefs_next(node_idx[0], node_idx[1])

            commit_to_next_lattice(eqn, node_idx[0]+1, node_idx[1], left_coefs[0, 0])
            commit_to_next_lattice(eqn, node_idx[0], node_idx[1]+1, left_coefs[1, 0])
            commit_to_next_lattice(eqn, node_idx[0], node_idx[1], left_coefs[2, 0])
            commit_to_next_lattice(eqn, node_idx[0]-1, node_idx[1], left_coefs[3, 0])
            commit_to_next_lattice(eqn, node_idx[0], node_idx[1]-1, left_coefs[4, 0])
        return next_lattice_coefs, bias

    def step(self):
        lattice_coefs, bias = self._build_next_system()
        solution = np.linalg.solve(lattice_coefs, bias)
        self.history[self.current_time] = np.copy(self.lattice)
        self.current_time += 1
        for node_idx, node in enumerate(self.nodes):
            node_value = solution[node_idx, 0]
            self.lattice[node[0], node[1]] = node_value if node_value >= 0 else 0
        return self.lattice


if __name__ == "__main__":
    if len(sys.argv) < 7:
        print(f"Usage: {sys.argv[0]} <k_1> <k_2> <lambda> <emitter-strength> <emitter1-x> <e
        exit()
    _, k_1, k_2, l, emitter_strength, emitter1x, emitter1y, emitter2x, emitter2y, n, frames

    k_1 = float(k_1)
    k_2 = float(k_2)
    l = float(l)
    phi = float(emitter_strength)
    emitter1x = int(emitter1x)
    emitter1y = int(emitter1y)
    emitter2x = int(emitter2x)
    emitter2y = int(emitter2y)
    n = int(n)
    frames = int(frames)

    # k_1, k_2 = [0.5, 0.3]
    # k_1, k_2 = [0.5, 0.5]
    # l = 0.4                       # lambda
```

```
# n = 15                          # number of nodes

x, y = [(0, 10), (0, 10)]       # observed area
t_step = 1                      # time step
u_crit = 100                    # critical concentration
# phi = 5                       # emitters strength
emitters = [(emitter1x, emitter1y), (emitter2x, emitter2y)]  # emitters location at node
initial = lambda x, y: 0

solver = DiffusionDiffEquationSolver(k_1, k_2, l,
                                     n, x, y,
                                     t_step, u_crit,
                                     emitters, phi,
                                     initial)

frames_n = frames
for i in range(frames_n):
    solver.step()

print(f"Solved system for {frames_n} steps")
print("System data:")
print(f"    emitters  = {emitters}")
print(f"    time step = {solver.t_step}")
print(f"    x step    = {solver.x_step}")
print(f"    y step    = {solver.y_step}")
print(f"    k_1  = {solver.k_1}")
print(f"    k_2  = {solver.k_2}")
print(f"    l    = {solver.l}")
print(f"    n    = {solver.n}")

max_concentration = np.max(np.array([solver.history[i] for i in solver.history]))

animate_method = 'all'

def animate_surface():
    plt.rcParams["figure.figsize"] = [7.50, 7.50]
    plt.rcParams["figure.autolayout"] = True

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.view_init(30, 0)
    title = ax.set_title('Diffusion Process')
    data = solver.history[0]
    X, Y = np.meshgrid(solver.nodesx, solver.nodesy)
    ax.set_zlim(-0.1, max_concentration)
    ax.plot_surface(X, Y, data,
                    cmap='Reds', edgecolor='black')

    def update(i):
        print(f"\r  rendering frame {i}", end='')
```

```
        ax.view_init(elev=30., azim=i + 5)
        data = solver.history[i]
        ax.plot_surface(X, Y, data,
                        cmap='Reds', edgecolor='black')
        title.set_text(f"Diffusion process, time index = {i} with {k_1=}, {k_2=}, {l=}")

    ani = animation.FuncAnimation(fig, update, frames_n,
                                  interval=200)
    ani.save(f"surface-animation-{n}-{int(time.time())}.mp4")

def animate_heatmap():
    plt.rcParams["figure.figsize"] = [7.50, 7.50]
    plt.rcParams["figure.autolayout"] = True

    fig = plt.figure()
    kwargs = {
        'square': True,
        'vmin': 0,
        'cmap': 'hot',
        'robust': True,
    }

    def init():
        sns.heatmap(solver.history[0], cbar=True, **kwargs)

    def animate(i):
        print(f"\r  rendering frame {i}", end='')
        data = solver.history[i]
        sns.heatmap(data, cbar=False, **kwargs)

    anim = animation.FuncAnimation(fig, animate, init_func=init,
                                   frames=frames_n, repeat=False)
    anim.save(f"heatmap-animation-{n}-{int(time.time())}.mp4")

def animate_all():
    plt.rcParams["figure.figsize"] = [15.0, 7.50]
    plt.rcParams["figure.autolayout"] = True

    fig = plt.figure()
    ax_surf = fig.add_subplot(1, 2, 1, projection=Axes3D.name)
    ax_surf.view_init(30, 0)
    ax_heat = fig.add_subplot(1, 2, 2)

    title = fig.suptitle('Diffusion process')
    ax_surf.view_init(30, 0)

    data = solver.history[0]
    X, Y = np.meshgrid(solver.nodesx, solver.nodesy)
    ax_surf.plot_surface(X, Y, data,
                         cmap='Reds', edgecolor='black')
```

```
    heat_kwargs = {
        'vmin': 0,
        'vmax': max_concentration,
        'cmap': 'hot',
    }

    def init():
        ax_heat.pcolormesh(X, Y, solver.history[0], **heat_kwargs)

    def animate(i):
        print(f"\r  rendering frame {i}", end='')
        data = solver.history[i]

        ax_heat.pcolormesh(X, Y, data, **heat_kwargs)

        ax_surf.view_init(elev=30., azim=i + 5)
        ax_surf.plot_surface(X, Y, data,
                             cmap='Reds', edgecolor='black')
        title.set_text(f"Diffusion process, time index = {i} with {k_1=}, {k_2=}, {l=}")

    anim = animation.FuncAnimation(fig, animate, init_func=init,
                                   frames=frames_n, repeat=False)
    anim.save(f"animation-{n}-{int(time.time())}.mp4")

def plot_all(moment):
    fig = plt.figure()
    ax_surf = fig.add_subplot(1, 2, 1, projection=Axes3D.name)
    ax_surf.view_init(30, 0)
    ax_heat = fig.add_subplot(1, 2, 2)

    heat_kwargs = {
        'vmin': 0,
        'vmax': max_concentration,
        'cmap': 'hot',
    }

    title = fig.suptitle('Diffusion process')
    title.set_text(f"Diffusion process, time index = {i} with {k_1=}, {k_2=}, {l=}")
    ax_surf.view_init(30, 0)

    data = solver.history[moment]
    X, Y = np.meshgrid(solver.nodesx, solver.nodesy)
    ax_surf.plot_surface(X, Y, data,
                         cmap='Reds', edgecolor='black')

    ax_heat.pcolormesh(X, Y, solver.history[0], **heat_kwargs)
    fig.savefig(f"plot({moment})-{n}-{int(time.time())}.mp4")

plot_all(int(frames_n * 0.95))
```

```
if animate_method == 'heatmap':
    print('Heat animation process')
    animate_heatmap()
if animate_method == 'surface':
    print('Surface animation process')
    animate_surface()
if animate_method == 'all':
    print('Surface and heatmap animation process')
    animate_all()
```