ИУ5-32Б
Ушаков Л. И.

Отчёт по ДЗ по курсу ПиК ЯП

**Задание**

1. Выберите язык программирования (который Вы ранее не изучали) и (1) напишите по нему реферат с примерами кода или (2) реализуйте на нем небольшой проект (с детальным текстовым описанием).
2. Реферат (проект) может быть посвящен отдельному аспекту (аспектам) языка или содержать решение какой-либо задачи на этом языке.
3. Необходимо установить на свой компьютер компилятор (интерпретатор, транспилятор) этого языка и произвольную среду разработки.
4. В случае написания реферата необходимо разработать и откомпилировать примеры кода (или модифицировать стандартные примеры).
5. В случае создания проекта необходимо детально комментировать код.
6. При написании реферата (создании проекта) необходимо изучить и корректно использовать особенности парадигмы языка и основных конструкций данного языка.
7. Приветствуется написание черновика статьи по результатам выполнения ДЗ. Черновик статьи может быть подготовлен группой студентов, которые исследовали один и тот же аспект в нескольких языках или решили одинаковую задачу на нескольких языках.

**Описание и листинг кода**

**Постановка задачи:** написать программу, вычисляющую двумерное электростатическое поле системы, путём решения уравнения Пуассона с помощью метода конечных элементов (МКЭ) и непрерывного метода Галёркина.

**Описание реализации:** Проект моделирует двумерное электростатическое поле с проводниками с использованием метода конечных элементов (МКЭ): сначала создаётся ограниченная триангуляция Делоне для квадратной области с учётом границ проводников. Для каждого треугольника вычисляются локальные матрицы жёсткости с учётом диэлектрической проницаемости среды. Затем собирается глобальная разреженная матрица системы и решается линейная система для нахождения потенциалов в узлах.

Полученные значения потенциалов используются для вычисления модуля электрического поля, который визуализируется через цветовую карту INFERNO на треугольной сетке. В качестве графической библиотеки выбран Nannou - кроссплатформенный фреймворк на Rust для работы с визуализацией и интерактивной графикой

**Листинг кода:**

```
use nannou::prelude::*;
use nannou::draw::mesh::{
Mesh
};

use russell_lab::math::{
SQRT_3
};
use spade::{
AngleLimit,
ConstrainedDelaunayTriangulation,
Point2,
RefinementParameters,
Triangulation
};

use russell_sparse::prelude::*;
use russell_sparse::StrError;
use russell_lab::Vector;

use std::collections::HashSet;
use std::f32;

use colorous::INFERNO;
```

```
struct Conductor {
id: usize,
node_ids: HashSet<usize>,
edges: HashSet<[usize; 2]>,
potential: Option<f64>,
total_charge: Option<f64>,
boundary: Box<dyn Fn(f64) -> (f64, f64)>
}

fn sample_parametric_closed<F>(
f: &F,
t0: f64,
t1: f64,
h_boundary: f64
) -> Vec<Point2<f64>>
where
F: Fn(f64) -> (f64, f64)
{
let mut points: Vec<Point2<f64>> = Vec::new();

let mut t = t0;
let (x0, y0) = f(t0);

let mut last = Point2::new(x0, y0);
points.push(last);

while t < t1 {
let dt = (h_boundary * 0.25 * 1e-2).min(1e-3) * (t1 - t0);
t = (t + dt).min(t1);

let (x, y) = f(t);
let p = Point2::new(x, y);

let dx = p.x - last.x;
let dy = p.y - last.y;

let dist2 = dx*dx + dy*dy;

if dist2 >= h_boundary * h_boundary {
points.push(p);
last = p;
}
}

points
```

```
}

fn charge_density(x: f64, y: f64) -> f64 {
0.0
}

fn relative_permittivity(x: f64, y: f64) -> f64 {
if (0.4 <= x && x <= 0.6) && (0.47 <= y && y <= 0.53) {
let t = (y - 0.47) / (0.53 - 0.47);

let eps1 = 3.0;
let eps2 = 6.0;

return t * eps1 + (1.0 - t) * eps2;
}

1.0
}

fn export_mesh(cdt: &ConstrainedDelaunayTriangulation<Point2<f64>>) -> (Vec<[f64; 2]>,
Vec<[usize; 2]>, Vec<[usize; 3]>) {
use std::collections::HashMap;

let mut nodes = Vec::new();
let mut handle_to_index = HashMap::<_, usize>::new();

for (i, v) in cdt.vertices().enumerate() {
handle_to_index.insert(v, i);
let p = v.position();
nodes.push([p.x, p.y]);
}

let mut elements = Vec::new();
for face in cdt.inner_faces() {
let verts = face.vertices();

elements.push([
handle_to_index[&verts[0]],
handle_to_index[&verts[1]],
handle_to_index[&verts[2]]
]);
}

let mut edges = Vec::new();
for edge in cdt.undirected_edges() {
```

```
let verts = edge.vertices();

edges.push([
handle_to_index[&verts[0]],
handle_to_index[&verts[1]]
])
}

(nodes, edges, elements)
}

fn add_unit_square_boundary(
cdt: &mut ConstrainedDelaunayTriangulation<Point2<f64>>,
h: f64,
) {
let n = (1.0 / h).ceil().max(1.0) as usize;
let step = 1.0 / n as f64;

let mut pts: Vec<Point2<f64>> = Vec::new();

for i in 0..=n {
let x = i as f64 * step;
pts.push(Point2::new(x, 0.0));
}

for i in 1..n {
let y = i as f64 * step;
pts.push(Point2::new(1.0, y));
}

for i in (0..=n).rev() {
let x = i as f64 * step;
pts.push(Point2::new(x, 1.0));
}

for i in (1..n).rev() {
let y = i as f64 * step;
pts.push(Point2::new(0.0, y));
}

let mut handles = Vec::with_capacity(pts.len());
for p in pts {
let v = cdt.insert(p).unwrap();
handles.push(v);
}
```

```
let m = handles.len();
for i in 0..m {
let a = handles[i];
let b = handles[(i + 1) % m];
cdt.add_constraint(a, b);
}
}

fn get_triangulation(triangles_estimate: usize, conductors: &mut [Conductor]) -> (Vec<[f64; 2]>,
Vec<[usize; 2]>, Vec<[usize; 3]>) {
let mut cdt: ConstrainedDelaunayTriangulation<Point2<f64>> =
ConstrainedDelaunayTriangulation::new();

let mut constraints: Vec<Vec<Point2<f64>>> = Vec::new();

let h: f64 = (4.0 / SQRT_3 / triangles_estimate as f64).sqrt();
add_unit_square_boundary(&mut cdt, h);

for cond in conductors.iter() {
let constraint = sample_parametric_closed(&cond.boundary, 0.0, 1.0, h);
constraints.push(constraint);
}

for constraint in &constraints {
let mut vertex_handles: Vec<_> = vec![];
for point in constraint {
let v = cdt.insert(*point).unwrap();
vertex_handles.push(v);
}

let n = vertex_handles.len();
for i in 0..n {
cdt.add_constraint(vertex_handles[i], vertex_handles[(i + 1) % n]);
}
}

let params = RefinementParameters::new()
.with_max_allowed_area(1.0 / (triangles_estimate as f64))
.with_angle_limit(AngleLimit::from_deg(30.0))
.with_max_additional_vertices(3 * triangles_estimate)
.keep_constraint_edges();

let result = cdt.refine(params);
assert!(result.refinement_complete);

let (nodes, edges, elements) = export_mesh(&cdt);
```

```
for i in 0..conductors.len() {
for j in 0..nodes.len() {
for k in 0..constraints[i].len() {
if nodes[j][0] == constraints[i][k].x && nodes[j][1] == constraints[i][k].y {
conductors[i].node_ids.insert(j);
}
}
}

for j in 0..edges.len() {
for k in 0..constraints[i].len() {
let c1x = constraints[i][k].x;
let c1y = constraints[i][k].y;

let c2x = constraints[i][(k+1) % constraints[i].len()].x;
let c2y = constraints[i][(k+1) % constraints[i].len()].y;

let e1x = nodes[edges[j][0]][0];
let e1y = nodes[edges[j][0]][1];
let e2x = nodes[edges[j][1]][0];
let e2y = nodes[edges[j][1]][1];

let same =
(c1x == e1x && c1y == e1y && c2x == e2x && c2y == e2y) ||
(c1x == e2x && c1y == e2y && c2x == e1x && c2y == e1y);



if same {
conductors[i].edges.insert(edges[j]);
}
}
}
}

(nodes, edges, elements)
}

fn assemble_global(
n_dofs: usize,
elements: &[([usize; 3], Vec<Vec<f64>>)],
floating_conductors: &Vec<&Conductor>
) -> Result<CooMatrix, StrError> {
let mut n_lambda = 0;
let m = floating_conductors.len();
```

```
for cond in floating_conductors.iter() {
n_lambda += cond.node_ids.len();
}

let nnz_estimate = elements.len() * 9 + 2 * m * n_lambda + 2 * n_lambda * n_dofs;

let mut coo = CooMatrix::new(n_dofs + n_lambda + m, n_dofs + n_lambda + m, nnz_estimate,
Sym::No)?;

for (local_indices, local_stiffness_matrix) in elements {
for i in 0..3 {
for j in 0..3 {
let global_i = local_indices[i];
let global_j = local_indices[j];

coo.put(global_i, global_j, local_stiffness_matrix[i][j])?;
}
}
}

Ok(coo)
}

fn is_on_unit_square_boundary(x: f64, y: f64) -> bool {
let eps = 1e-12;

(x < eps) || (y < eps) || (x > 1.0 - eps) || (y > 1.0 - eps)
}

fn compute_nodal_coefficients(
triangulation: (&Vec<[f64; 2]>, &Vec<[usize; 3]>),
conductors: &Vec<Conductor>
) -> Vec<f64> {
let (nodes, elements) = triangulation;

let mut local_contributions: Vec<([usize; 3], Vec<Vec<f64>>)> = Vec::new();
let mut f: Vec<f64> = vec![0.0; nodes.len()];

let eps0 = 8.854e-12;

for local_indices in elements {
let mut local_stiffness_matrix: Vec<Vec<f64>> = Vec::new();
let (i, j, k) = (
nodes[local_indices[0]],
nodes[local_indices[1]],
```

```
nodes[local_indices[2]],
);

let b = [j[1] - k[1], k[1] - i[1], i[1] - j[1]];
let c = [k[0] - j[0], i[0] - k[0], j[0] - i[0]];

let two_area = i[0] * b[0] + j[0] * b[1] + k[0] * b[2];
let area = two_area.abs() / 2.0;

let area_eps = 1e-18;
if !area.is_finite() || area < area_eps {
eprintln!(
"degenerate tri: area={:e} i={:?} j={:?} k={:?} idx={:?}",
area, i, j, k, local_indices
);
continue;
}

for m in 0..3 {
let mut stiffness_matrix_row: Vec<f64> = Vec::with_capacity(3);

for n in 0..3 {
let cx = (i[0] + j[0] + k[0]) / 3.0;
let cy = (i[1] + j[1] + k[1]) / 3.0;

let eps = relative_permittivity(cx, cy);

let k_mn = eps * eps0 * (b[m] * b[n] + c[m] * c[n]) / (4.0 * area);
stiffness_matrix_row.push(k_mn);
}
local_stiffness_matrix.push(stiffness_matrix_row);
}

local_contributions.push((*local_indices, local_stiffness_matrix));

let rho = [
charge_density(i[0], i[1]),
charge_density(j[0], j[1]),
charge_density(k[0], k[1]),
];
let rho_avg = (rho[0] + rho[1] + rho[2]) / 3.0;

let local_f = [rho_avg * area / 3.0; 3];

f[local_indices[0]] += local_f[0];
```

```
f[local_indices[1]] += local_f[1];
f[local_indices[2]] += local_f[2];
}
let n = nodes.len();
let mut is_dirichlet_bc = vec![false; n];
let mut dirichlet_bc_values = vec![0.0; n];

let mut dirichelt_bc_nodes_count: usize = 0;

for i in 0..n {
if is_on_unit_square_boundary(nodes[i][0], nodes[i][1]) {
dirichelt_bc_nodes_count += 1;
is_dirichlet_bc[i] = true;
dirichlet_bc_values[i] = 0.0;
}
}

let mut floating_conductors: Vec<&Conductor> = Vec::new();

for cond in conductors {
if let Some(potential) = cond.potential {
for &nid in &cond.node_ids {
is_dirichlet_bc[nid] = true;
dirichlet_bc_values[nid] = potential;
dirichelt_bc_nodes_count += 1;
}
} else {
floating_conductors.push(cond);
}
}

let mut edge_index = 0;

for cond in floating_conductors.iter() {
for edge_indices in cond.edges.iter() {
let lx = nodes[edge_indices[0]][0] - nodes[edge_indices[1]][0];
let ly = nodes[edge_indices[0]][1] - nodes[edge_indices[1]][1];

let edge_length = (lx*lx + ly*ly).sqrt();

edge_index += 1;
}
}

let global_stiffness_matrix = assemble_global(nodes.len(), &local_contributions,
&floating_conductors)
```

```
.expect("assemble_global failed");

let ii = global_stiffness_matrix.get_row_indices();
let jj = global_stiffness_matrix.get_col_indices();
let vv = global_stiffness_matrix.get_values();

let nnz_estimate = local_contributions.len() * 9 + dirichelt_bc_nodes_count;
let mut global_stiffness_matrix_with_dirichelet_bc = CooMatrix::new(n, n, nnz_estimate,
Sym::No).unwrap();

for p in 0..vv.len() {
let i = ii[p] as usize;
let j = jj[p] as usize;
let v = vv[p];
if is_dirichlet_bc[i] {
continue;
}
if is_dirichlet_bc[j] {
f[i] -= v * dirichlet_bc_values[j];
continue;
}
let _ = global_stiffness_matrix_with_dirichelet_bc.put(i, j, v);
}
for i in 0..n {
if is_dirichlet_bc[i] {
let _ = global_stiffness_matrix_with_dirichelet_bc.put(i, i, 1.0);
f[i] = dirichlet_bc_values[i];
}
}

let mut solver = LinSolver::new(Genie::Klu)
.expect("failed to create linear solver");

solver.actual
.factorize(&global_stiffness_matrix_with_dirichelet_bc, None)
.expect("factorization failed");

let rhs = Vector::from(&f);
let mut u = Vector::filled(f.len(), 0.0);

solver.actual
.solve(&mut u, &rhs, false)
.expect("solve failed");

u.as_data().to_vec()
```

```
}



fn local_to_view(_app: &App, point: [f64; 2]) -> [f32; 2] {
let win = _app.window_rect();
let w = win.w() * ((point[0] - 0.5) as f32);
let h = win.h() * ((point[1] - 0.5) as f32);

[w, h]
}

struct Model {
mesh: Mesh
}

fn main() {
nannou::app(model)
.event(event)
.simple_window(view)
.size(1024, 1024)
.run();
}

fn model(_app: &App) -> Model {
let mut conductors: Vec<Conductor> = vec![];

fn rect_parametric(t: f64, a: [f64; 2], b: [f64; 2], c: [f64; 2]) -> (f64, f64) {
let u = 4.0 * t;

if 0.0 <= u && u <= 1.0 {
let s = u;

let x = c[0] - a[0] - b[0] + 2.0 * s * a[0];
let y = c[1] - a[1] - b[1] + 2.0 * s * a[1];

return (x,y);
} else if 1.0 <= u && u <= 2.0 {
let s = u - 1.0;

let x = c[0] + a[0] - b[0] + 2.0 * s * b[0];
let y = c[1] + a[1] - b[1] + 2.0 * s * b[1];

return (x,y);
} else if 2.0 <= u && u <= 3.0 {
let s = u - 2.0;
```

```
let x = c[0] + a[0] + b[0] - 2.0 * s * a[0];
let y = c[1] + a[1] + b[1] - 2.0 * s * a[1];

return (x,y);
} else if 3.0 <= u && u <= 4.0 {
let s = u - 3.0;

let x = c[0] - a[0] + b[0] - 2.0 * s * b[0];
let y = c[1] - a[1] + b[1] - 2.0 * s * b[1];

return (x,y);
} else {
(0.0, 0.0)
}
}

conductors.push(Conductor {
id: 1,
node_ids: HashSet::new(),
potential: Some(600.0),
total_charge: None,
boundary: Box::new(|t: f64| {
let a = [0.25, 0.0];
let b = [0.0, 0.0005];
let c = [0.5, 0.45];

rect_parametric(t, a, b, c)
}),
edges: HashSet::new()
});

conductors.push(Conductor {
id: 2,
node_ids: HashSet::new(),
potential: Some(-600.0),
total_charge: None,
boundary: Box::new(|t: f64| {
let a = [0.25, 0.0];
let b = [0.0, 0.0005];
let c = [0.5, 0.55];

rect_parametric(t, a, b, c)
}),
edges: HashSet::new()
});
```

```
let (nodes, edges, elements) = get_triangulation(1000000, &mut conductors);

let nodal_coefficients = compute_nodal_coefficients((&nodes, &elements), &conductors);
let min_val = nodal_coefficients.iter().cloned().fold(f64::INFINITY, f64::min) as f32;
let max_val = nodal_coefficients.iter().cloned().fold(f64::NEG_INFINITY, f64::max) as f32;

println!("{}", min_val);
println!("{}", max_val);

let mut points: Vec<Vec3> = Vec::with_capacity(elements.len() * 3);
let mut indices: Vec<u32> = Vec::with_capacity(elements.len() * 3);
let mut colors: Vec<LinSrgba> = Vec::with_capacity(elements.len() * 3);
let mut tex_coords: Vec<Vec2> = Vec::with_capacity(elements.len() * 3);

let mut next_index: u32 = 0;
for [i0, i1, i2] in elements.iter().copied() {
let p0 = local_to_view(_app, nodes[i0]);
let p1 = local_to_view(_app, nodes[i1]);
let p2 = local_to_view(_app, nodes[i2]);

let d = (p1[0] - p0[0]) * (p2[1] - p0[1]) - (p2[0] - p0[0]) * (p1[1] - p0[1]);

let dz2 = (nodal_coefficients[i1] - nodal_coefficients[i0]) as f32;
let dz3 = (nodal_coefficients[i2] - nodal_coefficients[i0]) as f32;

let e_x = (dz2 * (p2[1] - p0[1]) - dz3 * (p1[1] - p0[1])) / d;
let e_y = (dz3 * (p1[0] - p0[0]) - dz2 * (p2[0] - p0[0])) / d;

let e = (e_x*e_x + e_y*e_y).sqrt();

fn gradient(x: f32, k: f32) -> nannou::color::LinSrgba {
let k = if k.is_finite() && k > 0.0 { k } else { 1.0 };
let x = if x.is_finite() && x > 0.0 { x } else { 0.0 };

let t = (x / k).asinh();
let t = t / (1.0 + t);
let c = INFERNO.eval_continuous(t as f64);

fn srgb_to_linear(c: f32) -> f32 {
if c <= 0.04045 {
c / 12.92
} else {
((c + 0.055) / 1.055).powf(2.4)
}
```

```
}

LinSrgba::new(
srgb_to_linear(c.r as f32 / 255.0),
srgb_to_linear(c.g as f32 / 255.0),
srgb_to_linear(c.b as f32 / 255.0),
1.0
)
}

let color = gradient(e, 10.0);

points.push(pt3(p0[0], p0[1], 0.0));
points.push(pt3(p1[0], p1[1], 0.0));
points.push(pt3(p2[0], p2[1], 0.0));

colors.push(color);
colors.push(color);
colors.push(color);

let t = pt2(0.0, 0.0);
tex_coords.push(t);
tex_coords.push(t);
tex_coords.push(t);

indices.push(next_index);
indices.push(next_index + 1);
indices.push(next_index + 2);

next_index += 3;
}

let mut mesh = Mesh::default();
mesh.extend_from_slices(&points, &indices, &colors, &tex_coords);

Model { mesh }
}

fn event(_app: &App, _model: &mut Model, _event: Event) {

}

fn view(_app: &App, _model: &Model, _frame: Frame) {
let draw = _app.draw();
```

```
draw.background().color(WHITE);

draw.mesh()
.indexed_colored(
_model.mesh.points().iter().copied().zip(_model.mesh.colors().iter().copied()),
_model.mesh.indices().iter().map(|&i| i as usize),
);
draw.to_frame(_app, &_frame).unwrap();
}
```

**Скриншот работы программы:**

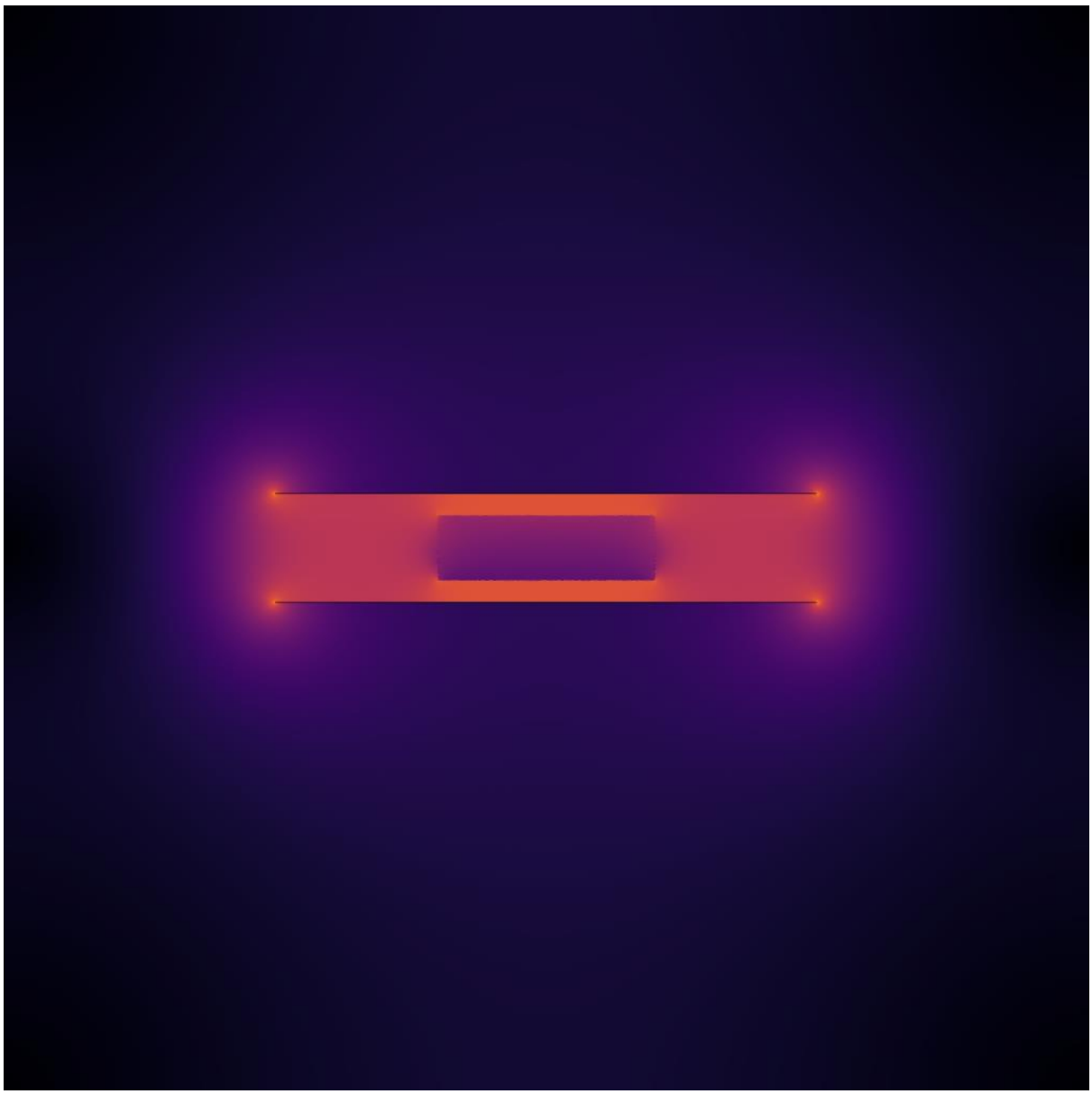

Рис. 1 (Электростатическое поле подключенного конденсатора с прямоугольным диэлектриком внутри. Диэлектрик линейно меняет диэлектрическую проницаемость в зависимости от расстояния до одной из обкладок)

**Вывод:**

В результате работы над проектом мною был выучен язык программирования Rust, а также был изучен МКЭ для решения ДУ в частных переменных.