MEEN 644 - Homework 4

Logan Harbour

March 3, 2019

Problem statement

Consider a thin copper square plate of dimensions 0.5 m × 0.5 m. The temperature of the west and south edges are maintained at 50 °C and the north edge is maintained at 100 °C. The east edge is insulated. Using finite volume method, write a program to predict the steady-state temperature solution.

(a) **(35 points)** Set the over relaxation factor α from 1.00 to 1.40 in steps of 0.05 to identify α_{opt}. Plot the number of iterations required for convergence for each α.

(b) **(15 points)** Solve the same problem using $21^2, 25^2, 31^2$, and 41^2 CVs, respectively. Plot the temperature at the center of the plate (0.25 m, 0.25 m) vs CVs.

(c) **(10 points)** Plot the steady state temperature contour in the 2D domain with the 41^2 CV solution.

Preliminaries

Two-dimensional heat conduction

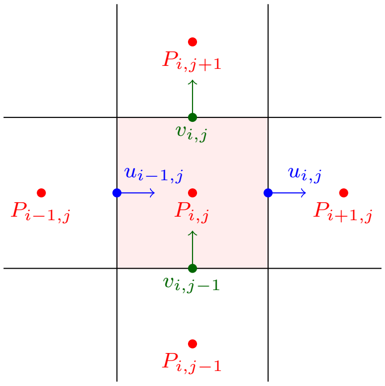
With two-dimensional heat conduction with constant material properties, insulation on the right and prescribed temperatures on all other sides, we have the PDE

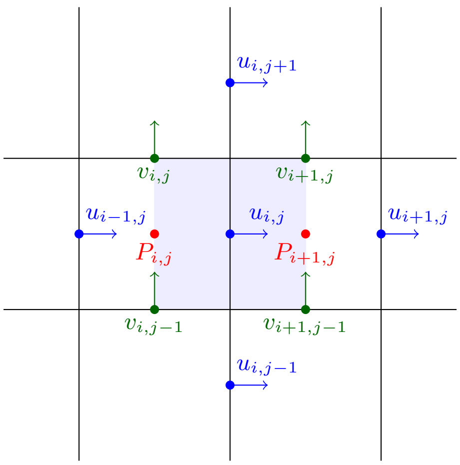
$$\begin{cases} k\frac{\partial^2 T}{\partial x^2} + k\frac{\partial^2 T}{\partial y^2} = 0\,, \\ T(x,0) = T_B\,, \\ T(0,y) = T_L\,, \\ T(0,L_y) = T_T\,, \\ -k\frac{\partial T}{\partial x}\Big|_{x=L_x} = 0\,, \end{cases} \tag{1}$$

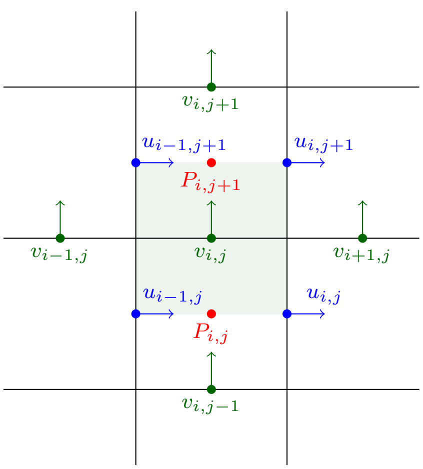
where

$$T_B \equiv 50\ ^\circ\text{C}\,, \qquad T_L \equiv 50\ ^\circ\text{C}\,, \qquad T_T \equiv 100\ ^\circ\text{C}\,.$$
$$k \equiv 386\ \text{W/m}\ ^\circ\text{C}\,, \qquad L_x \equiv 0.5\ \text{m}\,, \qquad L_y \equiv 0.5\ \text{m}\,.$$

Control volume equations







Velocity update

Define the Pechlet number on each boundary of a control volume $c_{i,j}$ as

$$P_b^{c_{i,j}} = \frac{F_b^{c_{i,j}}}{D_b^{c_{i,j}}}\,, \quad \text{where} \quad b = [n, e, s, w] \quad \text{and} \quad c = [u, v]\,, \tag{2}$$

where

$$
\begin{aligned}
D_n^{c_{i,j}} &= \frac{\Delta x \mu}{\Delta y}\,, && (3a)\\
D_e^{c_{i,j}} &= \frac{\Delta y \mu}{\Delta x}\,, && (3b)\\
D_s^{c_{i,j}} &= \frac{\Delta x \mu}{\Delta y}\,, && (3c)\\
D_w^{c_{i,j}} &= \frac{\Delta y \mu}{\Delta x}\,. && (3d)
\end{aligned}
$$

u-velocity update

Integrating the x-momentum equation (with the guessed variables and neglecting the $\frac{\partial v^*}{\partial x}$ term) an internal u-velocity control volume and using the power-law scheme, we obtain

$$
a_p^{u_{i,j}} u_{i,j}^* = a_n^{u_{i,j}} u_{i,j+1}^* + a_e^{u_{i,j}} u_{i+1,j}^* + a_s^{u_{i,j}} u_{i,j-1}^* + a_w^{u_{i,j}} u_{i-1,j}^* + \Delta y^{u_{i,j}} (p_{i,j}^* - p_{i+1,j}^*)\,, \tag{4}
$$

where

$$
\begin{aligned}
a_n^{u_{i,j}} &= D_n^{u_{i,j}} \max\left[0, (1-0.1|P_n^{u_{i,j}}|)^5\right] + \max\left[-F_n^{u_{i,j}}, 0\right]\,, && (5a)\\
a_e^{u_{i,j}} &= D_e^{u_{i,j}} \max\left[0, (1-0.1|P_e^{u_{i,j}}|)^5\right] + \max\left[-F_e^{u_{i,j}}, 0\right]\,, && (5b)\\
a_s^{u_{i,j}} &= D_s^{u_{i,j}} \max\left[0, (1-0.1|P_s^{u_{i,j}}|)^5\right] + \max\left[F_s^{u_{i,j}}, 0\right]\,, && (5c)\\
a_w^{u_{i,j}} &= D_w^{u_{i,j}} \max\left[0, (1-0.1|P_w^{u_{i,j}}|)^5\right] + \max\left[F_w^{u_{i,j}}, 0\right]\,, && (5d)\\
a_p^{u_{i,j}} &= a_n^{u_{i,j}} + a_e^{u_{i,j}} + a_s^{u_{i,j}} + a_w^{u_{i,j}}\,, && (5e)
\end{aligned}
$$

and

$$
\begin{aligned}
F_n^{u_{i,j}} &= \frac{1}{2}\rho \Delta x^{u_{i,j}} \left(v_{i,j} + v_{i+1,j}\right)\,, && (6a)\\
F_e^{u_{i,j}} &= \frac{1}{2}\rho \Delta y^{u_{i,j}} \left(u_{i,j} + u_{i+1,j}\right)\,, && (6b)\\
F_s^{u_{i,j}} &= \frac{1}{2}\rho \Delta x^{u_{i,j}} \left(v_{i,j-1} + v_{i+1,j-1}\right)\,, && (6c)\\
F_w^{u_{i,j}} &= \frac{1}{2}\rho \Delta y^{u_{i,j}} \left(u_{i-1,j} + u_{i,j}\right)\,. && (6d)
\end{aligned}
$$

There exist the following manipulations for the boundary control volumes:

- On the left and right boundaries:

$$
D_n^{u_{i,j}} = \frac{3\Delta x \mu}{2\Delta y}\,, \quad i = 1, M_x^u - 1\,, \quad 0 < j < M_y^u\,, \tag{7}
$$

$$
D_s^{u_{i,j}} = \frac{3\Delta x \mu}{2\Delta y}\,, \quad i = 1, M_x^u - 1\,, \quad 0 < j < M_y^u\,, \tag{8}
$$

- On the right boundary:

$$F_n^{u_{M_x^u-1,j}} = \frac{\rho\Delta x}{4}\left(2v_{M_y^v-2,j} + 3v_{M_y^v-1,j} + v_{M_y^v,j}\right)\,, \quad 0 < j < M_y^u\,, \tag{9}$$

$$F_s^{u_{M_x^u-1,j}} = \frac{\rho\Delta x}{4}\left(2v_{M_y^v-2,j-1} + 3v_{M_y^v-1,j-1} + v_{M_y^v,j-1}\right)\,, \quad 0 < j < M_y^u\,, \tag{10}$$

$$F_e^{u_{M_x^u-1,1}} = \frac{\rho\Delta y}{2}\left(u_{M_x^u,0} + u_{M_x^u,1makmak}\right)\,, \tag{11}$$

$$F_e^{u_{M_x^u-1,M_y^u-1}} = \frac{\rho\Delta y}{2}\left(u_{M_x^u,M_y^u} + u_{M_x^u,M_y^u-1}\right)\,, \tag{12}$$

$$F_e^{u_{M_x^u-1,j}} = \rho\Delta y u_{M_x^u,j}\,, \quad 1 < j < M_y^u - 1\,, \tag{13}$$

- On the left boundary:

$$F_n^{u_{1,j}} = \frac{\rho\Delta x}{4}\left(v_{0,j} + 2v_{1,j} + 3v_{2,j}\right)\,, \quad 0 < j < M_y^u\,, \tag{14}$$

$$F_s^{u_{1,j}} = \frac{\rho\Delta x}{4}\left(v_{0,j-1} + 2v_{1,j-1} + 3v_{2,j-1}\right)\,, \quad 0 < j < M_y^u\,, \tag{15}$$

$$F_w^{u_{1,1}} = \frac{\rho\Delta y}{2}\left(u_{0,0} + u_{0,1}\right)\,, \tag{16}$$

$$F_w^{u_{1,M_y^u-1}} = \frac{\rho\Delta y}{2}\left(u_{0,M_y^u-1} + u_{0,M_y^u}\right)\,, \tag{17}$$

$$F_w^{u_{1,j}} = \rho\Delta y u_{0,j}\,, \quad 1 < M_y^u - 1\,, \tag{18}$$

v-velocity update

Integrating the x-momentum equation (with the guessed variables and neglecting the $\frac{\partial vu^*}{\partial y}$ term) an internal v-velocity control volume and using the power-law scheme, we obtain

$$a_p^{v_{i,j}} v_{i,j}^* = a_n^{v_{i,j}} v_{i,j+1}^* + a_e^{v_{i,j}} v_{i+1,j}^* + a_s^{v_{i,j}} v_{i,j-1}^* + a_w^{v_{i,j}} v_{i-1,j}^* + \Delta x^{u_{i,j}}\left(p_{i,j}^* - p_{i,j+1}^*\right)\,, \tag{19}$$

where

$$a_n^{v_{i,j}} = D_n^{v_{i,j}} \max\left[0, (1 - 0.1|P_n^{v_{i,j}}|)^5\right] + \max\left[-F_n^{v_{i,j}}, 0\right]\,, \tag{20a}$$

$$a_e^{v_{i,j}} = D_e^{v_{i,j}} \max\left[0, (1 - 0.1|P_e^{v_{i,j}}|)^5\right] + \max\left[-F_e^{v_{i,j}}, 0\right]\,, \tag{20b}$$

$$a_s^{v_{i,j}} = D_s^{v_{i,j}} \max\left[0, (1 - 0.1|P_s^{v_{i,j}}|)^5\right] + \max\left[F_s^{v_{i,j}}, 0\right]\,, \tag{20c}$$

$$a_w^{v_{i,j}} = D_w^{v_{i,j}} \max\left[0, (1 - 0.1|P_w^{v_{i,j}}|)^5\right] + \max\left[F_w^{v_{i,j}}, 0\right]\,, \tag{20d}$$

$$a_p^{v_{i,j}} = a_n^{v_{i,j}} + a_e^{v_{i,j}} + a_s^{v_{i,j}} + a_w^{v_{i,j}}\,, \tag{20e}$$

and

$$F_n^{v_{i,j}} = \frac{1}{2}\rho\Delta x^{v_{i,j}}\left(v_{i,j+1} + v_{i,j}\right)\,, \tag{21a}$$

$$F_e^{v_{i,j}} = \frac{1}{2}\rho\Delta y^{v_{i,j}}\left(u_{i,j} + u_{i,j+1}\right)\,, \tag{21b}$$

$$F_s^{v_{i,j}} = \frac{1}{2}\rho\Delta x^{v_{i,j}}\left(v_{i,j-1} + v_{i,j}\right)\,, \tag{21c}$$

$$F_w^{v_{i,j}} = \frac{1}{2}\rho\Delta y^{v_{i,j}}\left(u_{i-1,j} + u_{i-1,j+1}\right)\,. \tag{21d}$$

There exist the following manipulations for the boundary control volumes:

- On the top and bottom boundaries:

$$D_e^{u_{i,j}} = \frac{3\Delta y\mu}{2\Delta x}\,, \quad 0 < j < M_x^u\,, \quad j = 1, M_y^u - 1\,, \tag{22}$$

$$D_w^{u_{i,j}} = \frac{3\Delta y\mu}{2\Delta x}\,, \quad 0 < j < M_x^u\,, \quad j = 1, M_y^u - 1\,, \tag{23}$$

- On the top boundary:

$$F_w^{v_{i,M_y^v-1}} = \frac{\rho\Delta y}{4}\left(u_{i-1,M_y^u} + 2u_{i-1,M_y^u-1} + 3u_{i-1,M_y^u-2}\right)\,, \quad 0 < i < M_x^v\,, \tag{24}$$

$$F_e^{v_{i,M_y^v-1}} = \frac{\rho\Delta y}{4}\left(u_{i,M_y^u} + 2u_{i,M_y^u-1} + 3u_{i,M_y^u-2}\right)\,, \quad 0 < i < M_x^v\,, \tag{25}$$

$$F_n^{v_{0,M_y^v-1}} = \frac{\rho\Delta x}{2}\left(v_{0,M_y^v} + u_{1,M_y^v}\right)\,, \tag{26}$$

$$F_n^{v_{M_x^v-1,M_y^v-1}} = \frac{\rho\Delta x}{2}\left(v_{M_x^v-1,M_y^v} + v_{M_x^v,M_y^v}\right)\,, \tag{27}$$

$$F_n^{v_{i,M_y^v-1}} = \rho\Delta x v_{i,M_y^v}\,, \quad 1 < i < M_x^v - 1\,, \tag{28}$$

- On the bottom boundary:

$$F_w^{v_{i,1}} = \frac{\rho\Delta y}{4}\left(u_{i-1,0} + 2u_{i-1,1} + 3u_{i-1,2}\right)\,, \quad 0 < i < M_x^v\,, \tag{29}$$

$$F_e^{v_{i,1}} = \frac{\rho\Delta y}{4}\left(u_{i,0} + 2u_{i,1} + 3u_{i,2}\right)\,, \quad 0 < i < M_x^v\,, \tag{30}$$

$$F_s^{v_{0,1}} = \frac{\rho\Delta x}{2}\left(v_{0,0} + u_{1,0}\right)\,, \tag{31}$$

$$F_s^{v_{M_x^v-1,1}} = \frac{\rho\Delta x}{2}\left(v_{M_x^v-1,0} + v_{M_x^v,0}\right)\,, \tag{32}$$

$$F_s^{v_{i,1}} = \rho\Delta x v_{i,0}\,, \quad 1 < i < M_x^v - 1\,, \tag{33}$$

Solving methodology

Results

Code listing

For the implementation, we have the following files:

- `Makefile` – Allows for compiling the c++ project with `make`.
- `hwk4.cpp` – Contains the `main()` function that is required by C that runs the cases requested in this problem set.
- `Flow2D.h` / `Flow2D.cpp` – Contains the `Flow2D` class which is the solver for the 2D problem required in this homework.
- `Matrix.h` – Contains the `Matrix` class which provides storage for a matrix with various standard matrix operations.
- `TriDiagonal.h` – Contains the `TriDiagonal` class which provides storage for a tri-diagonal matrix including the TDMA solver found in the member function `solveTDMA()`.
- `plots.py` - Produces the plots in this report.

Makefile

```
src = $(wildcard *.cpp)
obj = $(src:.cpp=.o)
CXXFLAGS = -std=c++14
CCFLAGS = $(CXXFLAGS)

hwk-opt: $(obj)
	clang++ -o $@ $^

.PHONY: clean
clean:
	rm -f $(obj) hwk-opt
```

hwk4.cpp

```
#include "Flow2D.h"
#include <boost/format.hpp>
#include <map>
#include <sstream>

int
main()
{
  double Re = 1000;
  double Lx = 0.1;
  double Ly = 0.1;
  double mu = 0.001002;
  double rho = 998.3;
  double bc_val = Re * mu / (rho * Lx);
  BoundaryCondition u_BC(bc_val, 0, bc_val, 0);
  BoundaryCondition v_BC(0, 0, 0, 0);

  Flow2D problem(5, 5, Lx, Ly, u_BC, v_BC, rho, mu);
  problem.solve();
}
```

Flow2D.h

```
#ifndef Flow2D_H
#define Flow2D_H

#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>

#include "Matrix.h"
#include "TriDiagonal.h"

template <typename T>
void
saveCSV(const std::vector<T> & v, std::string filename)
{
  std::ofstream f;
  f.open(filename);
  for (unsigned int i = 0; i < v.size(); ++i)
    f << std::scientific << v[i] << std::endl;
  f.close();
}

struct BoundaryCondition
```

```
{
  BoundaryCondition(double top, double right, double bottom, double left)
    : top(top), right(right), bottom(bottom), left(left)
  {
  }
  double top, right, bottom, left;
};

struct Coefficients
{
  double p, n, e, s, w, b;
};

struct MatrixCoefficients
{
  MatrixCoefficients(unsigned int Nx, unsigned int Ny) : vals(Nx, Ny) {}
  Coefficients & operator()(unsigned int i, unsigned int j) { return vals(i, j); }
  Matrix<Coefficients> vals;
};

/**
 * Solves a 2D heat conduction problem with dirichlet conditions on the top,
 * left, bottom and with a zero-flux condition on the right with Nx x Ny
 * internal control volumes.
 */
class Flow2D
{
public:
  Flow2D(unsigned int Nx,
         unsigned int Ny,
         double Lx,
         double Ly,
         BoundaryCondition u_BC,
         BoundaryCondition v_BC,
         double rho,
         double mu,
         unsigned int max_its = 1000);

  void solve();

  // See if this is solved/converged
  bool converged() { return (residuals.size() != 0 && residuals.size() != max_its); }

  // Get the residuals and number of iterations
  const std::vector<double> & getResiduals() const { return residuals; }
  unsigned int getNumIterations() { return residuals.size(); }

private:
  void fillBCs();
  void filluCoefficients();
  void fillvCoefficients();

  // Solve and sweep operations
  void solveu();
  void solvev();
  void solveuColumn(unsigned int i);
  void solveuRow(unsigned int j);
  void solvevColumn(unsigned int j);
  void solvevRow(unsigned int i);

protected:
  // Number of pressure CVs
  const unsigned int Nx, Ny;
  // Maximum nodal values
  const unsigned int M_x_u, M_y_u, M_x_v, M_y_v, M_x_p, M_y_p;

  // Geometry [m]
  const double Lx, Ly, dx, dy;
```

```
  // Boundary conditions
  const BoundaryCondition u_BC, v_BC;
  // Material properties
  const double rho, mu;
  // Coefficient matrices
  MatrixCoefficients a_u, a_v;

  // Maximum iterations
  const unsigned int max_its;
  // Relaxation coefficients
  const double w_u, w_v, alpha_p;

  // Velocity solutions
  Matrix<double> u, v;
  // Pressure solution
  Matrix<double> p;

  // Matrices and vectors for sweeping
  TriDiagonal<double> A_x_u, A_y_u, A_x_v, A_y_v;
  std::vector<double> b_x_u, b_y_u, b_x_v, b_y_v;

  // Residual for each iteration
  std::vector<double> residuals;
};

#endif /* Flow2D_H */
```

Flow2D.cpp

```
#include "Flow2D.h"

#include <cmath>

Flow2D::Flow2D(unsigned int Nx,
               unsigned int Ny,
               double Lx,
               double Ly,
               BoundaryCondition u_BC,
               BoundaryCondition v_BC,
               double rho,
               double mu,
               unsigned int max_its)
  : // Number of pressure CVs
    Nx(Nx),
    Ny(Ny),
    // Maixmum nodal values
    M_x_u(Nx),
    M_y_u(Ny + 1),
    M_x_v(Nx + 1),
    M_y_v(Ny),
    M_x_p(Nx + 1),
    M_y_p(Ny + 1),
    // Sizes
    Lx(Lx),
    Ly(Ly),
    dx(Lx / Nx),
    dy(Ly / Ny),
    // Boundary conditions
    u_BC(u_BC),
    v_BC(v_BC),
    // Material properties
    rho(rho),
    mu(mu),
    // Material properties in matrix form
    a_u(Nx + 1, Ny + 1),
```

```
    a_v(Nx + 1, Ny + 1),
    // Solver properties
    max_its(max_its),
    w_u(1 / 0.5),
    w_v(1 / 0.5),
    alpha_p(0.7),
    // Initialize coefficient matrices
    u(M_x_u + 1, M_y_u + 1),
    v(M_x_v + 1, M_y_v + 1),
    p(M_x_p + 1, M_y_p + 1),
    // Initialize sweeping matrices and vectors
    A_x_u(M_y_u - 1),
    A_y_u(M_x_u - 1),
    A_x_v(M_y_v - 1),
    A_y_v(M_x_v - 1),
    b_x_u(M_y_u - 1),
    b_y_u(M_x_u - 1),
    b_x_v(M_y_v - 1),
    b_y_v(M_x_v - 1)
{
}

void
Flow2D::solve()
{
  // Fill boundary conditions
  fillBCs();
  filluCoefficients();
  // fillvCoefficients();

  // solveu();
}

void
Flow2D::solveu()
{
  for (unsigned int j = 1; j < M_y_u; ++j)
    solveuRow(j);
}

void
Flow2D::fillBCs()
{
  u.setRow(0, u_BC.bottom);
  u.setRow(M_y_u, u_BC.top);
  u.setColumn(0, u_BC.left);
  u.setColumn(M_x_u, u_BC.right);
  v.setRow(0, v_BC.bottom);
  v.setRow(M_y_v, v_BC.top);
  v.setColumn(0, v_BC.left);
  v.setColumn(M_x_v, v_BC.right);
}

void
Flow2D::filluCoefficients()
{
  Coefficients D, F, P;

  for (unsigned int i = 1; i < M_x_u; ++i)
    for (unsigned int j = 1; j < M_y_u; ++j)
    {
      // Diffusion coefficient for left and right
      D.e = dy * mu / dx;
      D.w = dy * mu / dx;
      // Diffusion coefficient for top and bottom for internal cells
      if (i > 1 && i < M_x_u - 1)
      {
        D.n = 3 * dx * mu / (2 * dy);
```

```
        D.s = 3 * dx * mu / (2 * dy);
      }
      // Diffusion coefficient for top and bottom for left and right cells
      else
      {
        D.n = dx * mu / dy;
        D.s = dx * mu / dy;
      }

      // West flow rates
      if (i == 1)
        F.w = rho * dy * u(0, j);
      else
        F.w = rho * dy * (u(i - 1, j) + u(i, j)) / 2;
      // East flow rates
      if (i == M_x_u - 1)
        F.w = rho * dy * u(M_x_u, j);
      else
        F.w = rho * dy * (u(i - 1, j) + u(i, j)) / 2;
      // North and south flow rates on left boundary
      if (i == 1)
      {
        F.n = rho * dx * (v(0, j) + 2 * v(1, j) + 3 * v(2, j)) / 4;
        F.s = rho * dx * (v(0, j - 1) + 2 * v(1, j - 1) + 3 * v(2, j - 1)) / 4;
      }
      // North and south flow rates on right boundary
      else if (i == M_x_u - 1)
      {
        F.n = rho * dx * (2 * v(M_y_v - 2, j) + 3 * v(M_y_v - 1, j) + v(M_y_v, j)) / 4;
        F.s = rho * dx * (2 * v(M_y_v - 2, j - 1) + 3 * v(M_y_v - 1, j - 1) + v(M_y_v, j - 1)) / 4;
      }
      // North and south flow rates on the remainder
      else
      {
        F.n = rho * dx * (v(i, j) + v(i + 1, j)) / 2;
        F.s = rho * dx * (v(i, j - 1) + v(i + 1, j - 1)) / 2;
      }
      //
      // // Perchlet number
      // P.n = F.n / D.n;
      // P.e = F.e / D.e;
      // P.s = F.s / D.s;
      // P.w = F.w / D.w;
      //
      // // Fill coefficients
      // Coefficients & a = a_u(i, j);
      // a.n = D.n * std::fmax(0, std::pow(1 - 0.1 * std::fabs(P.n), 5)) + std::fmax(-F.n, 0);
      // a.e = D.e * std::fmax(0, std::pow(1 - 0.1 * std::fabs(P.e), 5)) + std::fmax(-F.e, 0);
      // a.s = D.s * std::fmax(0, std::pow(1 - 0.1 * std::fabs(P.s), 5)) + std::fmax(F.s, 0);
      // a.w = D.w * std::fmax(0, std::pow(1 - 0.1 * std::fabs(P.w), 5)) + std::fmax(F.w, 0);
      // a.p = a.n + a.e + a.s + a.w;
      // a.b = dy * (p(i, j) - p(i + 1, j));
    }
}

void
Flow2D::fillvCoefficients()
{
  Coefficients D, F, P;

  for (unsigned int i = 1; i < M_x_v; ++i)
    for (unsigned int j = 1; j < M_y_v; ++j)
    {
      // Diffusion coefficient for top and bottom
      D.n = dx * mu / dy;
      D.s = dx * mu / dy;
      // Diffusion coefficient for left and right for internal cells
      if (j > 1 && j < M_x_v - 1)
```

```
      {
        D.e = 3 * dy * mu / (2 * dx);
        D.w = 3 * dy * mu / (2 * dx);
      }
      // Diffusion coefficient for left and right for bottom and top cells
      else
      {
        D.e = dy * mu / dx;
        D.w = dy * mu / dx;
      }

      // North flow rates
      if (j == M_y_v - 1)
        F.n = rho * dx * v(i, M_y_v);
      else
        F.n = rho * dx * (v(i, j + 1) + v(i, j)) / 2;
      // South flow rates
      if (j == 1)
        F.s = rho * dx * v(i, 0);
      else
        F.s = rho * dx * (v(i, j - 1) + v(i, j)) / 2;
      // East and west flow rates on bottom boundary
      if (j == 1)
      {
        F.e = rho * dy * (u(i, 0) + 2 * u(i, 1) + 3 * u(i, 2)) / 4;
        F.w = rho * dy * (u(i - 1, 0) + 2 * u(i - 1, 1) + 3 * u(i - 1, 2)) / 4;
      }
      // East and west flow rates on top boundary
      else if (j == M_y_v - 1)
      {
        F.e = rho * dy * (u(i, M_y_u) + 2 * u(i, M_y_u - 1) + 3 * u(i, M_y_u - 2)) / 4;
        F.w = rho * dy * (u(i - 1, M_y_u) + 2 * u(i - 1, M_y_u - 1) + 3 * u(i - 1, M_y_u - 2)) / 4;
      }
      // East and west flow rates on the remainder
      else
      {
        F.e = rho * dy * (u(i, j) + u(i, j + 1)) / 2;
        F.w = rho * dy * (u(i - 1, j) + u(i - 1, j + 1)) / 2;
      }
    }
}

void
Flow2D::solveuRow(unsigned int j)
{
  std::cout << j << std::endl;
  for (unsigned int i = 1; i < M_x_u; ++i)
  {
    Coefficients & a = a_u(i, j);
    b_x_u[i - 1] = a.b + a.s * u(i, j - 1) + a.n * u(i, j + 1) + a.p * u(i, j) * (1 - w_u);
    if (i == 1)
    {
      A_x_u.setTopRow(a.p * w_u, -a.e);
      b_x_u[i - 1] += a.w * u(i - 1, j);
    }
    else if (i == M_x_u - 1)
    {
      A_x_u.setBottomRow(-a.w, a.p * w_u);
      b_x_u[i - 1] += a.e * u(i + 1, j);
    }
    else
      A_x_u.setMiddleRow(i - 1, -a.w, a.p * w_u, -a.e);
  }

  A_x_u.solveTDMA(b_x_u);
  for (unsigned int i = 1; i < M_x_u; ++i)
    u(i, j) = b_x_u[i - 1];
}
```

Matrix.h

```
#ifndef MATRIX
#define MATRIX

// #define NDEBUG
#include <cassert>
#include <vector>

/**
 * Class that holds a N x M matrix with common matrix operations.
 */
template <typename T>
class Matrix {
public:
  Matrix(unsigned int N, unsigned int M)
      : N(N), M(M), A(N, std::vector<T>(M)) {}

  // Const operator for getting the (i, j) element
  const T &operator()(unsigned int i, unsigned int j) const {
    assert(i < N && j < M);
    return A[i][j];
  }
  // Operator for getting the (i, j) element
  T &operator()(unsigned int i, unsigned int j) {
    assert(i < N && j < M);
    return A[i][j];
  }
  // Operator for setting the entire matrix to a value
  void operator=(T v) {
    for (unsigned int j = 0; j < M; ++j)
      setRow(j, v);
  }

  // Saves the matrix in csv format
  void save(const std::string filename, unsigned int precision = 12) const {
    std::ofstream f;
    f.open(filename);
    for (unsigned int j = 0; j < M; ++j) {
      for (unsigned int i = 0; i < N; ++i) {
        if (i > 0)
          f << ",";
        f << std::setprecision(precision) << A[i][j];
      }
      f << std::endl;
    }
    f.close();
  }

  // Set the j-th row to v
  void setRow(unsigned int j, T v) {
    assert(j < M);
    for (unsigned int i = 0; i < N; ++i)
      A[i][j] = v;
  }
  // Set the i-th column to v
  void setColumn(unsigned int i, T v) {
    assert(i < N);
    for (unsigned int j = 0; j < M; ++j)
      A[i][j] = v;
  }

  // Set the j-th row to vector v
  void setRow(unsigned int j, std::vector<T> &v) {
    assert(j < M && v.size() == N);
    for (unsigned int i = 0; i < N; ++i)
      A[i][j] = v[i];
```

```
  }
  // Set the i-th column to vector v
  void setColumn(unsigned int i, std::vector<T> &v) {
    assert(i < N && v.size() == M);
    for (unsigned int j = 0; j < M; ++j)
      A[i][j] = v[j];
  }

private:
  // The size of this matrix
  const unsigned int N, M;

  // Matrix storage
  std::vector<std::vector<T> > A;
};

#endif /* MATRIX_H */
```

TriDiagonal.h

```
#ifndef TRIDIAGONAL_H
#define TRIDIAGONAL_H

#define NDEBUG
#include <cassert>

/**
 * Class that holds a tri-diagonal matrix and is able to perform TDMA in place
 * with a given RHS.
 */
template <typename T>
class TriDiagonal {
public:
  TriDiagonal(unsigned int N, T v = 0)
      : N(N), A(N, v), B(N, v), C(N - 1, v) {}

  // Operator for setting the entire matrix to a value
  void operator=(TriDiagonal & from) {
    assert(from.getN() == N);
    A = from.getA();
    B = from.getB();
    C = from.getC();
  }

  // Gets the value of the (i, j) entry
  const T operator()(unsigned int i, unsigned int j) const {
    assert(i < N && j > i - 2 && j < i + 2);
    if (j == i - 1)
      return A[i];
    else if (j == i)
      return B[i];
    else if (j == i + 1)
      return C[i];
    else {
      std::cerr << "( " << i << "," << j << ") out of TriDiagonal system";
      std::terminate();
    }
  }

  // Adders for the top, middle, and bottom rows
  void addTopRow(T b, T c) {
    B[0] += b;
    C[0] += c;
  }
  void addMiddleRow(unsigned int i, T a, T b, T c) {
```

```cpp
    assert(i < N - 1 && i != 0);
    A[i] += a;
    B[i] += b;
    C[i] += c;
  }
  void addBottomRow(T a, T b) {
    A[N - 1] += a;
    B[N - 1] += b;
  }

  // Setters for the top, middle, and bottom rows
  void setTopRow(T b, T c) {
    B[0] = b;
    C[0] = c;
  }
  void setMiddleRow(unsigned int i, T a, T b, T c) {
    assert(i < N - 1 && i != 0);
    A[i] = a;
    B[i] = b;
    C[i] = c;
  }
  void setBottomRow(T a, T b) {
    A[N - 1] = a;
    B[N - 1] = b;
  }

  // Getters for the raw vectors
  const std::vector<T> &getA() const { return A; }
  const std::vector<T> &getB() const { return B; }
  const std::vector<T> &getC() const { return C; }

  // Getter for the size
  unsigned int getN() { return N; }

  // Saves the matrix in csv format
  void save(const std::string filename, unsigned int precision = 12) const {
    std::ofstream f;
    f.open(filename);
    for (unsigned int i = 0; i < N; ++i) {
        if (i > 0)
          f << std::setprecision(precision) << A[i] << ",";
        else
          f << "0" << ",";
        f << std::setprecision(precision) << B[i] << ",";
        if (i != N - 1)
          f << std::setprecision(precision) << C[i] << std::endl;
        else
          f << 0 << std::endl;
    }
    f.close();
  }

  // Solves the system Ax = d in place where d eventually stores the solution
  void solveTDMA(std::vector<T> &d) {
    // Forward sweep
    T tmp = 0;
    for (unsigned int i = 1; i < N; ++i) {
      tmp = A[i] / B[i - 1];
      B[i] -= tmp * C[i - 1];
      d[i] -= tmp * d[i - 1];
    }

    // Backward sweep
    d[N - 1] /= B[N - 1];
    for (unsigned int i = N - 2; i != std::numeric_limits<unsigned int>::max();
         --i) {
      d[i] -= C[i] * d[i + 1];
      d[i] /= B[i];
```

```cpp
    }
  }

protected:
  // Matrix size (N x N)
  unsigned int N;

  // Left/main/right diagonal storage
  std::vector<T> A, B, C;
};

#endif /* TRIDIAGONAL_H */
```

plots.py