MEEN 644 - Homework 4

Logan Harbour

March 2, 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

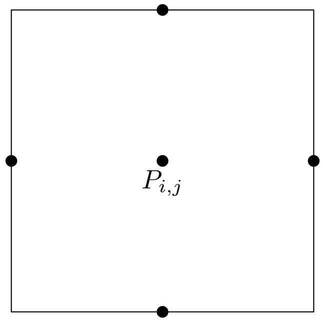
With two-dimensional heat conduction with constant material properties, insulation on the right and prescribed temperatures on all other sides, we have the PDE

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

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



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

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

int main() {

}
```

Flow2D.h

```cpp
#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;
};

/**
 * 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 k,
         double mu, double C_p, 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:
  double computeResidual() const;

  // Precompute operations
```

```
  void precomputeProperties();
  void precomputeColumn(unsigned int col);
  void precomputeRow(unsigned int row);

  // Solve and sweep operations
  void solveColumn(unsigned int col);
  void solveRow(unsigned int row);

protected:
  // Number of interior nodal points in the x and y-dimensions
  const unsigned int Nx, Ny;

  // Geometry [m]
  const double Lx, Ly, dx, dy;
  // Boundary conditions
  const BoundaryCondition u_BC, v_BC;
  // Material properties
  const double rho, k, mu, C_p;
  // Properties stored in matrix form
  Matrix<double> a_p, a_n, a_e, a_s, a_w;

  // Maximum iterations
  const unsigned int max_its;

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

  // Matrices for the TDMA solves
  TriDiagonal<double> A_x, A_y;
  // RHS/solution vector for the TDMA solves
  std::vector<double> b_x, b_y;
  // Residual for each iteration
  std::vector<double> residuals;
};

#endif /* Flow2D_H */
```

Flow2D.cpp

```
#include "Flow2D.h"

Flow2D::Flow2D(unsigned int Nx, unsigned int Ny, double Lx, double Ly,
               BoundaryCondition u_BC, BoundaryCondition v_BC, double rho,
               double k, double mu, double C_p,
               unsigned int max_its)
    : // Interior nodal points
      Nx(Nx), Ny(Ny), Lx(Lx), Ly(Ly), dx(Lx / Nx), dy(Ly / Ny),
      // Boundary conditions
      u_BC(u_BC), v_BC(v_BC),
      // Material properties
      rho(rho), k(k), mu(mu), C_p(mu),
      // Material properties in matrix form
      a_p(Nx, Ny), a_n(Nx, Ny), a_e(Nx, Ny), a_s(Nx, Ny), a_w(Nx, Ny),
      // Solver properties
      max_its(max_its),
      // Initialize matrices and vectors
      u(Nx - 1, Ny - 1), v(Nx - 1, Ny - 1), P(Nx + 1, Ny + 1), A_x(Nx), A_y(Ny),
      b_x(Nx), b_y(Ny) {}

void Flow2D::solve() {
  // Compute the a coefficients for each CV
  precomputeProperties();
  // Compute the unchanging LHS and RHS for each column
```

```cpp
  for (unsigned int i = 0; i < Nx; ++i)
    precomputeColumn(i);
  // Compute the unchanging LHS and RHS for each row
  for (unsigned int j = 0; j < Ny; ++j)
    precomputeRow(j);

  // Iterate and exit when complete
  for (unsigned int l = 1; l <= max_its; ++l) {
    // Sweep south to north
    for (int j = 0; j < Ny; ++j)
      solveRow(j);
    // Sweep west to east
    for (int i = 0; i < Nx; ++i)
      solveColumn(i);
    // Sweep north to south
    for (int j = Ny - 1; j >= 0; --j)
      solveRow(j);
    // Sweep east to west
    for (int i = Nx - 1; i >= 0; --i)
      solveColumn(i);

    // // Check for convergence and store residual
    // double R = computeResidual();
    // residuals.push_back(R);
    // if (R < tol) {
    //   std::cout << "Converged with " << l << " iterations" << std::endl;
    //   return;
    // }
  }

  std::cout << "Failed to converge in " << max_its << " iterations"
            << std::endl;
}

void Flow2D::precomputeProperties() {
  // // Set all neighbors to the default at first
  // a_n = k * dx / dy;
  // a_e = k * dy / dx;
  // a_s = k * dx / dy;
  // a_w = k * dy / dx;
  //
  // // Top Dirichlet
  // a_n.setRow(Ny - 1, 2 * k * dx / dy);
  // // Right Neumann
  // a_e.setColumn(Nx - 1, 0);
  // // Bottom Dirichlet
  // a_s.setRow(0, 2 * k * dx / dy);
  // // Left dirichlet
  // a_w.setColumn(0, 2 * k * dy / dx);
  //
  // // Center point
  // for (unsigned int i = 0; i < Nx; ++i)
  //   for (unsigned int j = 0; j < Ny; ++j)
  //     a_p(i, j) = a_n(i, j) + a_e(i, j) + a_s(i, j) + a_w(i, j);
}

void Flow2D::precomputeRow(unsigned int j) {
  // TriDiagonal<double> &A = pre_A_x[j];
  // std::vector<double> &b = pre_b_x[j];
  //
  // // First treat all as an internal volume
  // A.addTopRow(a_p(0, j) * w_inv, -a_e(0, j));
  // A.addBottomRow(-a_w(Nx - 1, j), a_p(Nx - 1, j) * w_inv);
  // for (unsigned int i = 1; i < Nx - 1; ++i)
  //   A.addMiddleRow(i, -a_w(i, j), a_p(i, j) * w_inv, -a_e(i, j));
  //
  // // Left Dirichlet
  // b[0] += T_L * a_w(0, j);
```

```
  // // Top dirichlet
  // if (j == Ny - 1)
  //   for (unsigned int i = 0; i < Nx; ++i)
  //     b[i] += T_T * a_n(i, j);
  // // Bottom dirichlet
  // else if (j == 0)
  //   for (unsigned int i = 0; i < Nx; ++i)
  //     b[i] += T_B * a_s(i, j);
}

void Flow2D::precomputeColumn(unsigned int i) {
  // TriDiagonal<double> &A = pre_A_y[i];
  // std::vector<double> &b = pre_b_y[i];
  //
  // // First treat all as an internal volume
  // A.addTopRow(a_p(i, 0) * w_inv, -a_n(i, 0));
  // A.addBottomRow(-a_s(i, Ny - 1), a_p(i, Ny - 1) * w_inv);
  // for (unsigned int j = 1; j < Ny - 1; ++j)
  //   A.addMiddleRow(j, -a_s(i, j), a_p(i, j) * w_inv, -a_n(i, j));
  //
  // // Left Dirichlet
  // if (i == 0)
  //   for (unsigned int j = 0; j < Ny; ++j)
  //     b[j] += T_L * a_w(i, j);
  // // Top Dirichlet
  // b[Ny - 1] += T_T * a_n(i, Ny - 1);
  // // Bottom Dirichlet
  // b[0] += T_B * a_s(i, 0);
}

void Flow2D::solveRow(unsigned int j) {
  // // Copy pre-filled Ax = b for this row
  // A_x = pre_A_x[j];
  // b_x = pre_b_x[j];
  //
  // // RHS contribution from volumes above and below
  // if (j > 0)
  //   for (unsigned int i = 0; i < Nx; ++i)
  //     b_x[i] += T(i, j - 1) * a_s(i, j);
  // if (j < Ny - 1)
  //   for (unsigned int i = 0; i < Nx; ++i)
  //     b_x[i] += T(i, j + 1) * a_n(i, j);
  //
  // // Relax, solve, and store solution (which is in b_x)
  // if (w_inv != 1)
  //   for (unsigned int i = 0; i < Nx; ++i)
  //     b_x[i] += (w_inv - 1.0) * a_p(i, j) * T(i, j);
  // A_x.solveTDMA(b_x);
  // T.setRow(j, b_x);
}

void Flow2D::solveColumn(unsigned int i) {
  //   // Copy pre-filled Ax = b for this row
  //   A_y = pre_A_y[i];
  //   b_y = pre_b_y[i];
  //
  //   // RHS contribution from volumes left and right
  //   if (i > 0)
  //     for (unsigned int j = 0; j < Ny; ++j)
  //       b_y[j] += T(i - 1, j) * a_w(i, j);
  //   if (i < Nx - 1)
  //     for (unsigned int j = 0; j < Ny; ++j)
  //       b_y[j] += T(i + 1, j) * a_e(i, j);
  //
  //   // Relax, solve, and store solution (which is in b_y)
  //   if (w_inv != 1)
  //     for (unsigned int j = 0; j < Ny; ++j)
  //       b_y[j] += (w_inv - 1.0) * a_p(i, j) * T(i, j);
```

```cpp
  //   A_y.solveTDMA(b_y);
  //   T.setColumn(i, b_y);
}

double Flow2D::computeResidual() const {
  // double R = 0.0, val = 0.0;
  // // Sum over all CVs
  // for (unsigned int i = 0; i < Nx; ++i)
  //   for (unsigned int j = 0; j < Ny; ++j) {
  //     // Main diagonal contribution and pre-computed RHS (from BC)
  //     val = a_p(i, j) * T(i, j) - pre_b_y[i][j];
  //     // Not on left boundary
  //     if (i > 0)
  //       val -= a_w(i, j) * T(i - 1, j);
  //     // Not on right boundary
  //     if (i < Nx - 1)
  //       val -= a_e(i, j) * T(i + 1, j);
  //     // Not on bottom boundary
  //     if (j > 0)
  //       val -= a_s(i, j) * T(i, j - 1);
  //     // Not top boundary
  //     if (j < Ny - 1)
  //       val -= a_n(i, j) * T(i, j + 1);
  //     R += std::abs(val);
  //   }
  // return R;
  return 0;
}
```

Matrix.h

```cpp
#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, T v = 0)
      : N(N), M(M), A(N, std::vector<T>(M, v)) {}

  // 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 && vs.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 && vs.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) {
    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;
  }

  // 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];
    }
  }

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