MECH 587 PROJECT 1

Presented to: Dr. Rajeev Jaiman

Christian Rowsell
40131393

Programming Assignment - 1

In this assignment, your final objective is to numerically solve the unsteady 2-D heat equation in a unit domain. The governing PDE in a domain Ω is given by,

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \quad \text{in } \Omega, \quad (1) \quad u = u_0 \text{ on } \partial\Omega_D, \quad (2)$$
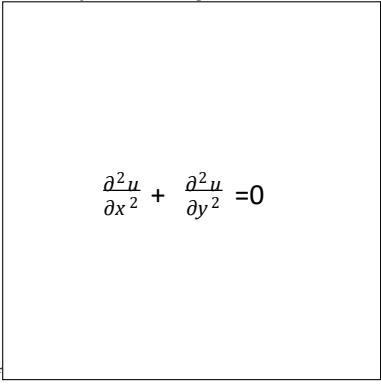
$$\frac{\partial u}{\partial n} = g \quad \text{on } \partial\Omega_N, \quad (3)$$

where $\partial\Omega_D$ and $\partial\Omega_N$ denote the regions of the boundary over which Dirichlet and Neumann boundary conditions are applied respectively. The final objective can be achieved by solving the following sub-problems.

1 Laplace's equation

In this section, you will solve the Laplace's equation over a unit domain given by

$$\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0 \text{ in } (0,1) \times (0,1),$$

$u = 1 - 6x^2 + x^4$

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

$u = y^4$

$u = 1 - 6y^2 + y^4$

$u = x^4$

Figure 1: Computational domain and boundary conditions for Laplace's equation

.

For the problem, your parameters are:

- Discretize the spatial derivatives using a 2nd order central differencing scheme. • Use a uniform and isotropic Cartesian grid, i.e. $\Delta x = constant$ and $\Delta x = \Delta y$.

- Solution on 3 different grids with 17, 33 and 65 grid points along each direction.

For the solution, your tasks are:

1. Write two functions/subroutines, which separately construct A and b corresponding to the Finite Difference Equations arising from the modified form of the PDE.

   Two functions were added to the provided example code in order to construct A and b utilizing the Finite Difference Equations. These are as follows.

   A –

```cpp
void computeMatrix(Matrix &M, const Grid &G)
{
            /*

*====================================================================
                  *Added boundary conditions based on whats given in steady

*====================================================================
            */

      unsigned long i,j;
      const double dx = G.dx();
      const double dy = G.dy();

      const double a = 1.0;
      const double b = 0.0;

      /*
       *a is 1.0 as M(i, j, 2) is the value that solves the point we are looking
at. Along the
       *boundary, this is 0. Seen in denotation for sparse matrix
       */

      for(i = 1; i < G.Nx()-1; i++)
          for(j = 1; j < G.Ny()-1; j++){
                M(i,j,0) = 1.0 / (dx * dx);
                M(i,j,1) = 1.0 / (dy * dy);
                M(i,j,2) = - 2.0 / (dx * dx) - 2.0 / (dy * dy);
                M(i,j,3) = 1.0 / (dy * dy);
                M(i,j,4) = 1.0 / (dx * dx);
          }

      for(i = 0; i < G.Nx(); i++)
          for(int t = 0; t < 5; t++){
                M(i,0,t) = (t == 2 ? a : b);   // Bottom boundary
                M(i,G.Ny()-1, t) = (t == 2 ? a : b);   // Top boundary
          }

      for(j = 0; j < G.Ny(); j++)
          for(int t = 0; t < 5; t++){
                M(0,j,t) = (t == 2 ? a : b);   // Left Boundary
                M(G.Nx()-1,j,t) = (t == 2 ? a : b);   // Right Boundary
          }
}
```

b –

```cpp
void computeDiffusion(Vector &R, const Vector &u, const Grid &G)
{
    // Added boundary condition calculated using Central Differencing method
    unsigned long i,j;
    unsigned long Nx = G.Nx();
    unsigned long Ny = G.Ny();
    double dx = G.dx();
    double dy = G.dy();

    for(i = 1; i < Nx-1; i++)
        for(j = 1; j < Ny-1; j++)
            // Computed using central differencing method for both X and Y
spatial derivative
            R(i,j) = (((u(i-1, j) - (2 * u(i, j)) + u(i+1, j)) / (dx * dx))
+ ((u(i, j-1) - (2 * u(i, j)) + u(i, j+1)) / (dy * dy)));

}
```

2. Compare the solution obtained with the exact solution given by $u_e(x,y) = x^4 + y^4 - 6x^2y^2$, by plotting the log-log plot of L_2 Norm of the error against grid-size, and compute the slope of the line obtained. Determine the order of accuracy of your solution.

L2 Norm Vs. Grid Size

3
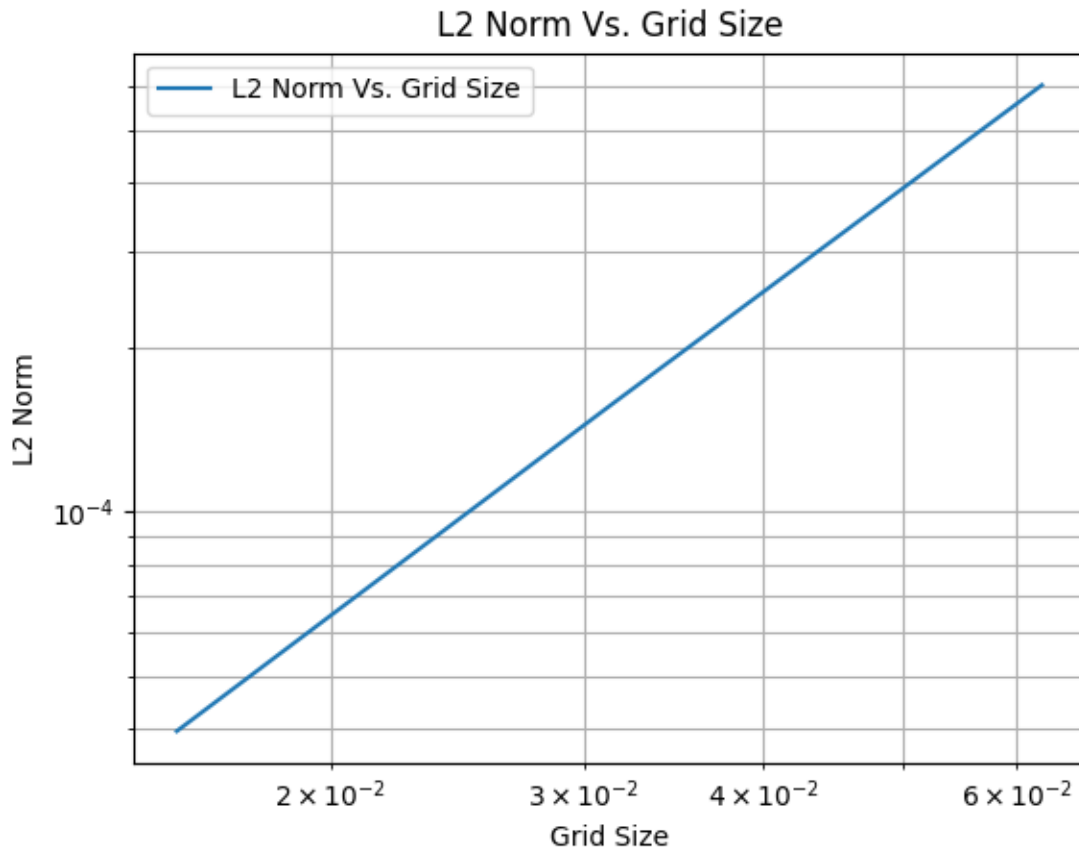
With this we get a slope of 1.96524. This makes sense, as the expected order of accuracy for the central difference method will be 2.

3. Tabulate the time taken to obtain a solution u against grid-size.

	17X17 Grid (289 Elements)	33X33 Grid (1089 Elements)	65X65 Grid (4225 Elements)
Total Time	2.016e-03	1.3619e-02	2.50909e-01

2 Unsteady Heat Equation

From section 1, you have constructed a code for applying the discrete Laplacian operator on a variable defined on Ω. In this section, you will use the ideas developed earlier to solve the Unsteady Heat Equation in an explicit and implicit manner up to a time level T. The PDE to solve in $(0,T) \times Ω$, now becomes

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$
$$u(x, y, 0) = \exp(-100(x^2 + y^2))$$

Using parameters and boundary conditions identical to those of the Laplace's equation, your tasks for the current problem are:

1. Derive the error-residual form ($A\delta u = b, u^{n+1} = u^n + \delta u$) for the Euler explicit method.

   Refer to the included hand written work at the end to see this.

2. The steady state occurs at time step n if $||u^{n+1} - u^n||_{L2} < 1e - 8$. Integrate the equation to steady state using (a) Euler explicit method and (b) Trapezoidal method. Write functions/subroutines which execute the time-integration process.

   a) Euler

```
void computeTransientMatrix(Matrix &M, const Grid &G, const double &dt)
{
            /*

*=================================================================
            *Added boundary conditions based on whats given in transient

*=================================================================
            */
  unsigned long i,j;
```

4

```cpp
    const double dx = G.dx();
    const double dy = G.dy();
    unsigned long Nx, Ny;
    Nx = G.Nx(), Ny = G.Ny();

    const double a = 1.0;
    const double b = 0.0;

    /*
     *a is 1.0 as M(i, j, 2) is the value that solves the point we are looking
    at. Along the
     *boundary, this is 0. Seen in denotation for sparse matrix
     */

    for(i = 1; i < Nx-1; i++)
        for(j = 1; j < Ny-1; j++) {
            /*
            Boundary conditions the same here, but with the difference
    (I/dt)
            I is 1 along the diagonal, and 0 everywhere else. Therefore, if
    the current position is not along the diagonal I/dt becomes 0,
            and if it is, it becomes 1/dt. Only M(i, j, 2) is along the
    diagonal.
            */
            M(i,j,0) = 0.0;
            M(i,j,1) = 0.0;
            M(i,j,2) = 1/dt;
            M(i,j,3) = 0.0;
            M(i,j,4) = 0.0;
        }

    for(i = 0; i < Nx; i++)
        for(int t = 0; t < 5; t++){
            M(i,0,t) = (t == 2 ? a : b);
            M(i,Ny-1, t) = (t == 2 ? a : b);
        }

    for(j = 0; j < Ny; j++)
        for(int t = 0; t < 5; t++){
            M(0,j,t) = (t == 2 ? a : b);
            M(Nx-1,j,t) = (t == 2 ? a : b);
        }
}
```

b) Trapezoidal

```cpp
void computeTransientMatrix(Matrix &M, const Grid &G, const double &dt)
{
            /*

*===================================================================
            *Added boundary conditions based on whats given in transient

*===================================================================
            */
    unsigned long i,j;
    const double dx = G.dx();
    const double dy = G.dy();
```

```cpp
    unsigned long Nx, Ny;
    Nx = G.Nx(), Ny = G.Ny();

    const double a = 1.0;
    const double b = 0.0;

    /*
     *a is 1.0 as M(i, j, 2) is the value that solves the point we are looking
    at. Along the
     *boundary, this is 0. Seen in denotation for sparse matrix
     */

    for(i = 1; i < Nx-1; i++)
        for(j = 1; j < Ny-1; j++) {
            /*
            Boundary conditions the same here, but with the difference (I/dt
    - 0.5 * A)
            I is 1 along the diagnol, and 0 everywhere else. Therefore if
    the current position is not along the diagnol I/dt becomes 0,
            and if it is, it becomes 1/dt. Only M(i, j, 2) is along the
    diagnol.
            */
            M(i,j,0) = -0.5 * (1 / (dx * dx));;
            M(i,j,1) = -0.5 * (1 / (dy * dy));
            M(i,j,2) = (1 / dt) - 0.5 * (- 2 / (dx * dx) - 2 / (dy * dy));
            M(i,j,3) = -0.5 * (1 / (dy * dy));
            M(i,j,4) = -0.5 * (1 / (dx * dx));
        }

    for(i = 0; i < Nx; i++)
        for(int t = 0; t < 5; t++){
            M(i,0,t) = (t == 2 ? a : b);
            M(i,Ny-1, t) = (t == 2 ? a : b);
        }

    for(j = 0; j < Ny; j++)
        for(int t = 0; t < 5; t++){
            M(0,j,t) = (t == 2 ? a : b);
            M(Nx-1,j,t) = (t == 2 ? a : b);
        }
}
```

3. For the Euler explicit method, obtain an upper bound on the time-step for which your solution does not diverge. Express your answer as $\frac{\Delta t}{\Delta x^2}$. Do the same for the trapezoidal scheme.

For the Euler method the timesteps 1e-1, 1e-2, 1e-3, 1e-4, and 1e-5 were tested for each grid size. For 289 elements, and 1089 elements, the highest timestep that the solution did not diverge was 1e-4, while for 4225 elements the highest timestep that the solution did not diverge was 1e-5. In this case $\frac{\Delta t}{\Delta x^2} = 0.0256\ for\ 17X17, \frac{\Delta t}{\Delta x^2} = 0.1024\ for\ 33X33, \frac{\Delta t}{\Delta x^2} = 0.04096\ for\ 65X65$

For trapezoidal timesteps of 1e-1, 5e-1, 1, 5, 10, and 20 were tested for each grid size. All of these converged, indicating that the trapezoidal method is unconditionally stable. To see more on this refer to the solution folders included. Information can be found in Console_Output.txt. For clarity,

the tabulated results for the Euler method can also be seen below. The Trapezoidal method was not tabulated, as the solution never diverged.

Euler Method

	17X17 Grid (289 Elements)	33X33 Grid (1089 Elements)	65X65 Grid (4225 Elements)
Δt	1e-4	1e-4	1e-5
$ΔX^2$	3.9062e-03	9.7656e-04	2.4414e-04
$\dfrac{Δt}{Δx^2}$	0.0256	0.1024	0.04096

4. Does the steady state solution match that obtained in section 1? Tabulate the values of $||u_{Laplace} - u_{Unsteady}||_{L2}$. Do you observe the initial condition influencing the steady state solution? In this regard, comment on the behaviour of the PDE?

In order to compare both the Euler and Trapezoidal methods, two different timesteps will be selected. For Trapezoidal, the timestep will be 1e-1 to stay in line with section 1, while for Euler the timestep will be 1e-5, as this is the highest timestep at which all 3 grid sizes converged. We will compare the L2 given for the unsteady solution with that of the steady solution.

Euler, dt=1e-5

	17X17 Grid (289 Elements)	33X33 Grid (1089 Elements)	65X65 Grid (4225 Elements)
$||U_{Laplace}-U_{Unsteady}||_{L2}$	5.05857622957e-05	5.050618046890002e-05	5.054692034315e-05

Trapezoidal, dt=1e-1

	17X17 Grid (289 Elements)	33X33 Grid (1089 Elements)	65X65 Grid (4225 Elements)
$||U_{Laplace}-U_{Unsteady}||_{L2}$	6.312318108599971e-06	8.250527426207599e-03	5.106255820088261e-02

7

The steady state solution matches that obtained in section 1 fairly well. For Euler it matches better than for Trapezoidal, most likely due to the small timestep used to ensure convergence. For Trapezoidal, the lower number of grid points matches better than the higher number of grid points. This is probably due to the high dt that is used for this simulation. A lower dt with 65 elements would probably match the steady state solution better. We can also plot the initial residual norm, to see if the same trends are followed. As this is the initial norm, the norm will be the same regardless of timestep and method used for time integration.

	17X17 Grid (289 Elements)	33X33 Grid (1089 Elements)	65X65 Grid (4225 Elements)
Initial Residual Norm	1.594832891076e+02	4.590769777501e+02	1.297070893422e+03

We can see that as the number of grid points increases, so too does the initial residual norm. For the trapezoidal method, this is interesting. As the number of grid points increases, so too does the initial, and final L2 residual norm. This indicates that the solution is not consistent, as the error increases as ΔX approaches 0. However, for the forward Euler method, we can see that the solution is quite consistent. At each different grid size, the final error was almost exactly the same, regardless of the fact that the initial residual norm varied at each grid size.

5. Tabulate the time taken to obtain a solution u against grid-size. Between solving the steady state equation, and time-integrating to steady state, which method was faster?

The following data was taken using dt=0.1 for Trapezoidal, and dt=1e-5 for Euler, in order to ensure convergence. The timestep also effects the solution time.

	17X17 Grid (289 Elements)	33X33 Grid (1089 Elements)	65X65 Grid (4225 Elements)
Total Time (s) - Euler	Steady State Time – 2.158e-03 Transient Time - 26.234879	Steady State Time – 1.3535e-02 Transient Time - 1.01611787e+02	Steady State Time – 2.61281e-01 Transient Time - 3.83921805e+02

Total Time (s) - Trapezoidal	Steady State Time – 2.016e-03	Steady State Time – 1.3619e-02	Steady State Time – 2.50909e-01
	Transient Time - 3.47684e-01	Transient Time - 8.406694076520e-03	Transient Time - 2.114762700000e+01

Referring to the above table, it becomes apparent that the transient solution takes longer to converge than the steady state solution does. This is especially true for the Euler method. This is because the timestep required to converge the forward Euler method is very small, and the number of iterations required for the transient solution is much higher. The difference is even greater with more grid points as well. The more grid points, the more equations that need to be solved per solution, which further increases the required time.

Some points to consider:

- The function/subroutine which constructs b, can compute the diffusion field for any variable defined over the domain Ω.

- Memory allocation for the construction of A and b has been implemented in the Matrix and Vector classes. This aspect has been addressed in the handout and the tutorial.

Equations to discretize

$$\frac{\partial u}{\partial t} = \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \text{ in } (0,1) \times (0,1)$$

Taking forward Euler Temporal discretization

$$\frac{I}{\Delta t} \left(u^{n+1} - u^n \right) = A u^n$$

where I is the identity matrix & A is some matrix representing central difference approximation of oure equation

We want to solve for u^{n+1}

$$\frac{I}{\Delta t} u^{n+1} = \underbrace{\left(A + \frac{I}{\Delta t} \right) u^n}_{b}$$

Apply Newton-Raphson iteration

$$\frac{I}{\Delta t} \left(u_k^{n+1} + \delta u_k^{n+1} \right) = b \qquad \text{where } k \text{ is the number of iterations}$$

$$\frac{I}{\Delta t} \left(\delta u_k^{n+1} \right) = b - \frac{I}{\Delta t} u_k^{n+1}$$

With convergence in 1 iteration, we take our initial guess of solution as the solution of the previous timestep.

$$u^{n+1} = u^n$$

The true solution after 1 iteration will be

$$u_1^{n+1} = u_0^{n+1} + \delta u_0^{n+1} = u^{n+1}$$

Substituting

$$\frac{I}{\Delta t} \left(\delta u_0^{n+1} \right) = b - \frac{I}{\Delta t} u_0^{n+1}$$

We now get

$$\frac{I}{\Delta t} \delta u^{n+1} = b - \frac{I}{\Delta t} u^n$$

$$u^{n+1} = u^n + \delta u^{n+1}$$

Substitute $b = \left(A + \frac{I}{\Delta t}\right) u^n$

$$\frac{I}{\Delta t}\left(\delta u^{n+1}\right) = \left(A + \frac{I}{\Delta t}\right) u^n - \frac{I}{\Delta t} u^n$$

finally

$$\frac{I}{\Delta t}\left(\delta u^{n+1}\right) = A u^n$$