CSE380 TOOLS AND TECHNIQUES FOR COMPUTATIONAL SCIENCE PROJECT 1 MODELING DOCUMENT

MOHAMMAD AFZAL SHADAB

Abstract. This is a *modeling document* for the application to solve the steady-state heat equation in one- and two-dimensions. The document highlights the governing equations, boundary conditions, numerical approximations, and high-level pseudocode implemented in the solver. Then we have described the files used in the software including C++, Python, shell scripts and text files. We have also outlined the procedures for building and running the codes and its testing. Next, we show the results of testing mainly verification, regression and runtime performance testing. We have used these libraries: `GRVY` for time-keeping and debugging, `MASA` for manufactured solutions, `bats` for designing regression tests. Finally, we have shown some results and compared them with analytical results.

1. Governing equations and boundary conditions

The steady-state heat equation with a constant coefficient in two dimensions is given by:

$$\boxed{-k\nabla^2 T(x,y) = q(x,y) \quad \forall \Omega \in [0,L] \times [0,H]} \tag{1}$$

where k is the thermal conductivity (W/K), $T(x,y)$ is the material temperature (K), $q(x,y)$ is a heat source term (W/m^2) and $\Omega \subset \mathbb{R}^2$ is the domain. This is Poisson equation which is a type of elliptic partial differential equations. This linear boundary value problem is subjected to either *Dirichlet boundary conditions* (T specified) following *maximum principle* or combinations of *Neumann* (∇T specified), *Dirichlet* and *Robin boundary conditions*. To begin with, let's specify

Date: November 25, 2020.

Dirichlet boundary conditions:

$$\begin{aligned} T(0,y) &= T_{analytical}(0,y) \\ T(L,y) &= T_{analytical}(L,y) \\ T(x,0) &= T_{analytical}(x,0) \\ T(x,H) &= T_{analytical}(x,H) \end{aligned} \tag{2}$$

where $T_{analytical}$ is evaluated using MASA.

2. Assumptions

We'll start with the assumptions for the derivation of equation 1 from the law of conservation of energy in Eulerian framework 3 [2]:

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot (\rho e \mathbf{u}) = \mathbf{T} : \mathbf{D} - \nabla \cdot \mathbf{q} + r \tag{3}$$

where e is internal energy per unit mass, $\mathbf{T} : \mathbf{D}$ is the strain heating, $\mathbf{q}$ is the heat flux and r is the volumetric source / sink.

- Continuum assumption
- Steady-state ($\partial(.)/\partial t = 0$)
- No advection $\mathbf{u} = \mathbf{0}$
- No source/sink term $r = 0$
- Validity of *Fourier's law of heat conduction*, i.e., $\mathbf{q} = -k\nabla T$
- No strain heating, i.e., $\mathbf{T} : \mathbf{D} = 0$, where $\mathbf{T}$ is the stress tensor and $\mathbf{D}$ is the deformation tensor
- Constant coefficient of thermal conductivity k

Moving towards the assumptions to make the numerical implementation easier:

- Square domain $L \equiv H$
- Uniform grid, i.e. $N_x = N_y = N$, where N_x and N_y are the cells in x and y directions respectively. So, $\Delta x = \Delta y = h$
- The Dirichlet boundary conditions are implemented in form of constant temperatures in the ghost cells adjacent to a corresponding boundary, i.e., $T_{0,1} = T_{analytical}(x_0, y_1)$

3. Nomenclature

For 1D, the mesh numbering is simple and straight forward, as shown in figure 1. The scheme is cell based, where cell centers (x_c, y_c) are referred. For 2D mesh, the situation is slightly sophisticated as two indices (i, j) come to picture correspondingly in x and y directions illustrated in figure 2. So, using a new numbering system for converting (i, j) into one index l which first spans x direction cells then marches in y direction.

$$
\begin{aligned}
l &= i + (j-1)N_x, \quad l \in \{1, 2, ..., N_x * N_y\} \\
l \% N_x &= i \qquad \text{(Remainder)} \\
l / N_x &= j - 1 \qquad \text{(Integer division)}
\end{aligned} \tag{4}
$$

The value at the centers of boundary cells are evaluated using MASA.

4. Numerical Methods

Finite difference approximation has been implemented considering the ease of implementation when compared with other discretization

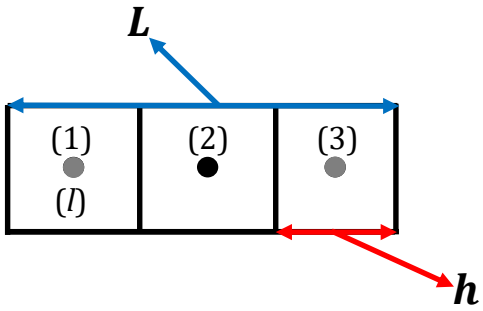


Figure 1. 1D mesh with 3 cells, grey dots which represent boundary cell centers, and l indexing

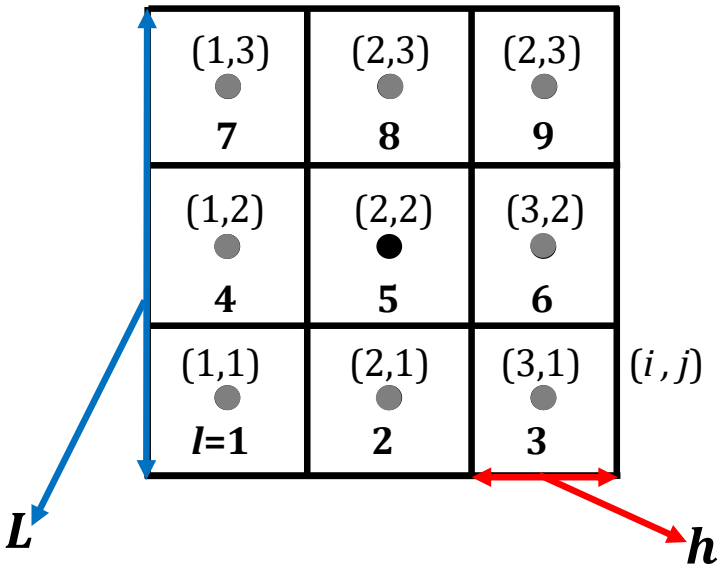


Figure 2. 2D 3×3 uniform mesh with boundary cells centers shown in grey, (i, j) indexing, and universal l indexing

methods such as finite volumes, finite element, etc [3]. Rewriting equation 1 in expanded form for 1D

$$-k\frac{\partial^2 T(x)}{\partial x^2} = q(x) \tag{5}$$

and for 2D

$$-k\left(\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2}\right) = q(x,y) \tag{6}$$

4.1. **Second order finite difference approximation.** Using the Taylor's expansion, the second order central difference approximation for second order derivative in x direction can be written

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} + \mathcal{O}(h^2) \tag{7}$$

where subscript i is the cell centered value of cell i.

4.1.1. *1D.* Equation 7 can be substituted into equation 5, to give

$$-k\left(\frac{T_{i+1} - 2T_{i,j} + T_{i-1}}{\Delta x^2}\right) + \mathcal{O}(h^2) = q_i \tag{8}$$

Inserting $\lambda = -k/\Delta x^2$ and dividing by λ, we get

$$T_{i+1} - 2T_i + T_{i-1} + \mathcal{O}(h^4) = q_i/\lambda \tag{9}$$

Neglecting the truncation error $\mathcal{O}(h^4)$ and writing in matrix form for a 3 cell grid shown in figure 1 after implementing the boundary conditions, we get

$$\begin{pmatrix} 1 & & \\ 1 & -2 & 1 \\ & & 1 \end{pmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} T_{1,MASA} \\ q_2/\lambda \\ T_{3,MASA} \end{bmatrix} \Rightarrow \mathbf{AT{=}q} \tag{10}$$

For a N-cell grid, the matrix equation takes the following form,

(11)

$$\mathbf{A} = \begin{pmatrix} 1 & & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & & 1 \end{pmatrix}, \mathbf{T} = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_{N-1} \\ T_N \end{bmatrix}, \mathbf{q} = \begin{bmatrix} T_{1,MASA} \\ q_2/\lambda \\ \vdots \\ q_{N-1}/\lambda \\ T_{N,MASA} \end{bmatrix}$$

So, the resulting matrix $\mathbf{A}$ is tridiagonal (3 diagonals).

$$\mathbf{A} = \begin{pmatrix} \times & & & & & \\ \times & \times & \times & & & \\ & \times & \times & \times & & \\ & & \ddots & \ddots & \ddots & \\ & & & \times & \times & \times \\ & & & & & \times \end{pmatrix} \tag{12}$$

4.1.2. *2D.* We can write

$$-k\left(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}\right) + \mathcal{O}(h^2) = q_{i,j} \tag{13}$$

where i and j are the positions of cell center in x and y directions. Using $\Delta x = \Delta y = h$ from assumptions and $-k/h^2 = \lambda$ we get

$$T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j} + \mathcal{O}(h^4) = q_{i,j}/\lambda \tag{14}$$

$$T_{k,k+1} + T_{k,k-1} + T_{k,k+N_x} + T_{k,k-N_x} - 4T_{k,k} + \mathcal{O}(h^4) = q_k/\lambda \tag{15}$$

Implementing boundary conditions, for a 5×5 uniform grid, the repeating interior block of $\mathbf{A}$ is given by,

$$\mathbf{A}_{rep} = \begin{bmatrix} & & 1 & & & & & & \\ 1 & \dots & 1 & -4 & 1 & \dots & 1 & & \\ & 1 & \dots & 1 & -4 & 1 & \dots & 1 & \\ & & 1 & \dots & 1 & -4 & 1 & \dots & 1 \\ & & & & & & 1 & & \end{bmatrix}$$

The rows corresponding to boundary entries only have a diagonal with value 1.

$$\mathbf{A} = \begin{bmatrix}
\times & & & & & & & & & & & & & & & & & & & & & & & & & \\
 & \times & & & & & & & & & & & & & & & & & & & & & & & & \\
 & & \times & & & & & & & & & & & & & & & & & & & & & & & \\
 & & & \times & & & & & & & & & & & & & & & & & & & & & & \\
 & & & & \times & & & & & & & & & & & & & & & & & & & & & \\
 & & & & & \times & & & & & & & & & & & & & & & & & & & & \\
\times & & & & & \times & \times & \times & & & & \times & & & & & & & & & & & & & & \\
 & \times & & & & \times & \times & \times & & & & & \times & & & & & & & & & & & & & \\
 & & \times & & & & \times & \times & \times & & & & & \times & & & & & & & & & & & & \\
 & & & & & & & & \times & & & & & & & & & & & & & & & & & \\
 & & & & & & & & & \times & & & & & & & & & & & & & & & & \\
 & & & & & \times & & & \times & \times & \times & & & & \times & & & & & & & & & & & \\
 & & & & & & \times & & & \times & \times & \times & & & & \times & & & & & & & & & & \\
 & & & & & & & \times & & & \times & \times & \times & & & & \times & & & & & & & & & \\
 & & & & & & & & & & & & & \times & & & & & & & & & & & & \\
 & & & & & & & & & & & & & & \times & & & & & & & & & & & \\
 & & & & & & & & & \times & & & & \times & \times & \times & & & & \times & & & & & & \\
 & & & & & & & & & & \times & & & & \times & \times & \times & & & & \times & & & & & \\
 & & & & & & & & & & & \times & & & & \times & \times & \times & & & & \times & & & & \\
 & & & & & & & & & & & & & & & & & & \times & & & & & & & \\
 & & & & & & & & & & & & & & & & & & & \times & & & & & & \\
 & & & & & & & & & & & & & & & & & & & & \times & & & & & \\
 & & & & & & & & & & & & & & & & & & & & & \times & & & & \\
 & & & & & & & & & & & & & & & & & & & & & & \times & & & \\
 & & & & & & & & & & & & & & & & & & & & & & & \ddots & &
\end{bmatrix}$$

So, we have

$$\Rightarrow \mathbf{AT} = \mathbf{q} \tag{16}$$

where

$\mathbf{q} = [T_{1,MASA}, T_{2,MASA} \ldots, q_{N+1}/\lambda, \ldots, q_{N*N-N}/\lambda, \ldots, T_{MASA}(x_{N*N-1}), T_{N*N,MASA}]^T$

$\mathbf{T} = [T_1, \ldots, T_{N*N}]^T$

For a general matrix,

$$A = \begin{cases}
A_{l-N_x,l} = \begin{cases} 1 & l/N_x = 0 \\ 0 & \text{else (bottom boundary)} \end{cases} \\
A_{l-1,l} = \begin{cases} 1 & l\%N_x > 1 \\ 0 & \text{else (left boundary)} \end{cases} \\
A_{l,l} = \begin{cases} -4 & \text{(interior)} \\ 1 & \text{else (boundary)} \end{cases} \\
A_{l+1,l} = \begin{cases} 1 & l\%N_x \ = 0 \\ 0 & \text{else (right boundary)} \end{cases} \\
A_{l-N_x,l} = \begin{cases} 1 & l/N_x = N_y - 1 \\ 0 & \text{else (top boundary)} \end{cases}
\end{cases} \tag{17}$$

4.2. **Fourth order finite difference approximation.** Using the Taylor's expansion, the fourth order central difference approximation for second order derivative in x direction can be written as:

$$\frac{\partial^2 T}{\partial x^2} = \frac{-T_{i+2} + 16T_{i+1} - 30T_i + 16T_{i-1} - T_{i-2}}{\Delta 12x^2} + \mathcal{O}(h^4) \tag{18}$$

where subscript i is the cell centered value of cell i.

4.2.1. *1D.* Equation 18 can be substituted into equation 5, to give

$$-k\left(\frac{-T_{i+2}+16T_{i+1}-30T_i+16T_{i-1}-T_{i-2}}{\Delta 12x^2}\right)+\mathcal{O}(h^4)=q_i \tag{19}$$

Inserting $\lambda=-k/\Delta x^2$ and dividing by λ, we get

$$-T_{i+2}+16T_{i+1}-30T_i+16T_{i-1}-T_{i-2}+\mathcal{O}(h^6)=q_i/\lambda \tag{20}$$

Neglecting the truncation error $\mathcal{O}(h^6)$ and writing in matrix form for an N cell grid shown in figure 1 after implementing the boundary conditions, we get

$$\Rightarrow \mathbf{AT}=\mathbf{q} \tag{21}$$

where

$$\mathbf{q}=[T_{1,MASA},T_{2,MASA},q_3/\lambda\ldots,q_{N-1}/\lambda,T_{N-1,MASA},T_{N,MASA}]^T$$
$$\mathbf{T}=[T(0),\ldots,T(N)]^T$$

$$\mathbf{A}=\begin{bmatrix}
1 & & & & & & & & & \\
 & 1 & & & & & & & & \\
-1 & 16 & -30 & 16 & -1 & & & & & \\
 & -1 & 16 & -30 & 16 & -1 & & & & \\
 & & -1 & 16 & -30 & 16 & -1 & & & \\
 & & & \ddots & \ddots & \ddots & \ddots & \ddots & & \\
 & & & & -1 & 16 & -30 & 16 & -1 & \\
 & & & & & -1 & 16 & -30 & 16 & -1 \\
 & & & & & & & & 1 & \\
 & & & & & & & & & 1
\end{bmatrix}$$

So, the resulting matrix $\mathbf{A}$ is pentadiagonal (5 diagonals)

$$\mathbf{A}=\begin{bmatrix}
\times & & & & & & & & & \\
 & \times & & & & & & & & \\
\times & \times & \times & \times & \times & & & & & \\
 & \times & \times & \times & \times & \times & & & & \\
 & & \times & \times & \times & \times & \times & & & \\
 & & & \ddots & \ddots & \ddots & \ddots & \ddots & & \\
 & & & & \times & \times & \times & \times & \times & \\
 & & & & & \times & \times & \times & \times & \times \\
 & & & & & & & & \times & \\
 & & & & & & & & & \times
\end{bmatrix}$$

4.2.2. *2D.* Equation 18 can be substituted into equation 7, to give

$$-k\left(\frac{-T_{i+2,j}+16T_{i+1,j}-30T_{i,j}+16T_{i-1,j}-T_{i-2,j}}{\Delta 12x^2} \quad + \right.$$
$$\left.\frac{-T_{i,j+2}+16T_{i,j+1}-30T_{i,j}+16T_{i,j-1}-T_{i,j-2}}{\Delta 12y^2}\right) \quad + \quad \mathcal{O}(h^4)=q_{i,j} \tag{22}$$

where i and j are the positions of cell center in x and y directions. Implementing, $\Delta x=\Delta y=h$ from assumptions and $-k/12h^2=\lambda$

$$-T_{i+2,j}+16T_{i+1,j}+16T_{i-1,j}-T_{i-2,j}-T_{i,j+2}+16T_{i,j+1}+16T_{i,j-1}-$$
$$T_{i,j-2}-60T_{i,j}+\mathcal{O}(h^6)=q_{i,j}/\lambda \tag{23}$$

Using k indexing for blocks in same row,

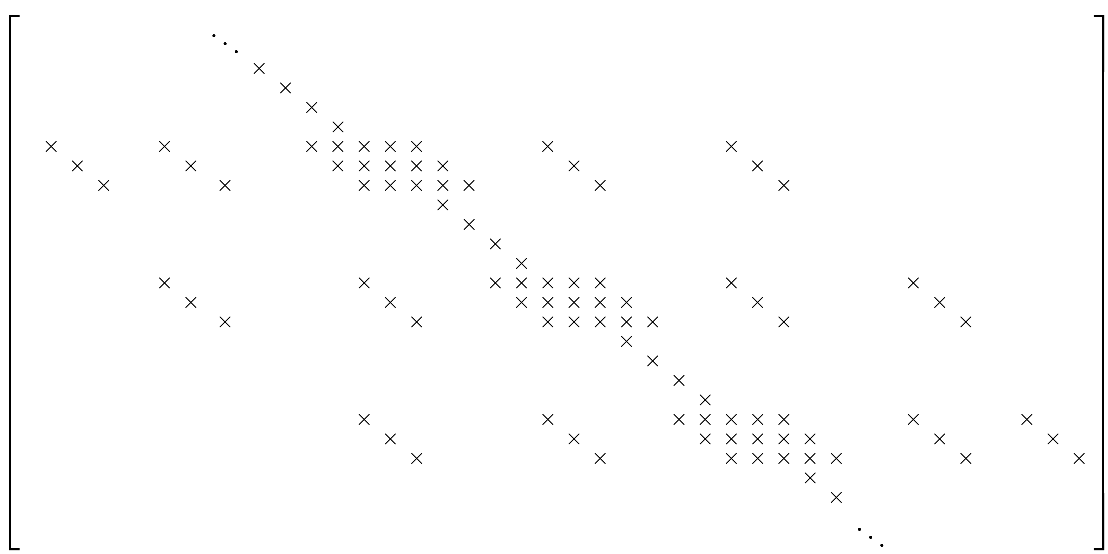
$$-T_{k+2,k}+16T_{k+1,k}+16T_{k-1,k}-T_{k-2,k}-T_{k+2N_x,k} \quad +16T_{k+N_x,k}+$$
$$16T_{k-N_x,k}-T_{k-2N_x,k}-60T_{k,k} \quad +\mathcal{O}(h^6)=q_k/\lambda \tag{24}$$

Neglecting the truncation error $\mathcal{O}(h^6)$ and writing in matrix form for 7×7 uniform grid after implementing the boundary conditions, we get

$$\mathbf{AT}=\mathbf{q}$$

where

$\mathbf{A}_{rep} =$



The repeating interior block of $\mathbf{A}$ is given by,

$$\mathbf{A} = \begin{bmatrix} & & & & & 1 & & & & & & & & \\ & & & & & & 1 & & & & & & & \\ 1 & \ldots & -16 & \ldots & 1 & -16 & 60 & -16 & 1 & \ldots & -16 & \ldots & 1 & \\ & 1 & \ldots & -16 & \ldots & 1 & -16 & 60 & -16 & 1 & \ldots & -16 & \ldots & 1 \\ & & 1 & \ldots & -16 & \ldots & 1 & -16 & 60 & -16 & 1 & \ldots & -16 & \ldots 1 \\ & & & & & & & & & 1 & & & & \\ & & & & & & & & & & 1 & & & \end{bmatrix}$$

So, the resulting matrix $\mathbf{A}$ is banded with 9 diagonals: principal diagonal, offdiagonals at distance 1, at distance N, and at distance $2N$ from principal diagonal.

5. Pseudocodes [1]

5.1. **Jacobi.** The algorithm is given below:

Input: $\mathbf{A}$= $A_{i,j}$, $\mathbf{B}$= B_j, $\mathbf{TO}$= TO_j=$\mathbf{T}^{(0)}$, tolerance TOL, max number of iterations N_{iter}

STEP 1 Set $k = 1$

STEP 2 While ($k \leq N$) do Steps 3-6

STEP 3 For $i = 1, 2, ..., n$

$$x_i = \frac{1}{a_{i,i}}\left[\sum_{j=1, j\neq i}^{n} (-A_{i,j}TO_j) + B_i\right]$$

STEP 4 If $||\mathbf{T} - \mathbf{TO}|| < TOL$ or $\frac{||\mathbf{T}^{(k)} - \mathbf{T}^{(k-1)}||}{||\mathbf{T}^{(k)}||} < TOL$, then OUTPUT($T_1, T_2, T_3, T_4, ..., T_n$); STOP

STEP 5 Set $k = k + 1$

STEP 6 For $i = 1, 2, ..., n$

Set $\mathbf{TO}_i = T_i$

STEP 7 OUTPUT ($T_1, T_2, T_3, T_4, ..., T_n$); STOP

5.2. **Gauss-Seidel.** The algorithm is given below:

Input: $\mathbf{A}$= $A_{i,j}$, $\mathbf{B}$= B_j, $\mathbf{TO}$= TO_j=$\mathbf{T}^{(0)}$, tolerance TOL, max number of iterations N_{iter}

STEP 1 Set $k = 1$

STEP 2 While ($k \leq N$) do Steps 3-6

STEP 3 For $i = 1, 2, ..., n$

$$x_i = \frac{1}{a_{i,i}}\left[-\sum_{j=1}^{i-1}(A_{i,j}T_j) - \sum_{j=i+1}^{n} (A_{i,j}TO_j) + B_i\right]$$

STEP 4 If $||\mathbf{T} - \mathbf{TO}|| < TOL$ or $\frac{||\mathbf{T}^{(k)} - \mathbf{T}^{(k-1)}||}{||\mathbf{T}^{(k)}||} < TOL$, then OUTPUT($T_1, T_2, T_3, T_4, ..., T_n$); STOP

STEP 5 Set $k = k + 1$

STEP 6 For $i = 1, 2, ..., n$
Set **T0**$_i = T_i$
STEP 7 OUTPUT $(T_1, T_2, T_3, T_4, ..., T_n)$; STOP

6. Memory requirement

Let's have a look at the memory requirement on case-by-case in the tables. Also, neglecting the elements at the top left and bottom right corners of the matrix **A**.

Table 1. Memory requirement for 1D, second order central difference

Variable	Dimension	Memory	Remarks
T	N	$8 \times N$	8-byte double precision, column vector
B	N	$8 \times N$	8-byte double precision, column vector
A	$N \times N$	$8 \times N \times N$	8-byte double precision square matrix
N	1	4×1	4-byte single precision, scalar
L	1	8×1	8-byte double precision, scalar
Δx	1	8×1	8-byte double precision, scalar
k	1	8×1	8-byte double precision, scalar
Total		$8N^2 + 16N + 28$	

Table 2. Memory requirement for 2D, second order central difference

Variable	Dimension	Memory	Remarks
T	N^2	$8 \times N^2$	8-byte double precision, column vector
B	N^2	$8 \times N^2$	8-byte double precision, column vector
A	$N^2 \times N^2$	$8 \times N^2 \times N^2$	8-byte double precision square matrix
N	1	4×1	4-byte single precision, scalar
$L \equiv H$	1	8×1	8-byte double precision, scalar
$\Delta x = \Delta y = h$	1	8×1	8-byte double precision, scalar
k	1	8×1	8-byte double precision, scalar
Total		$8N^4 + 16N^2 + 28$	

TABLE 3. Memory requirement for 1D, fourth order central difference

Variable	Dimension	Memory	Remarks
T	N	$8 \times N$	8-byte double precision, column vector
B	N	$8 \times N$	8-byte double precision, column vector
A	$N \times N$	$8 \times N \times N$	8-byte double precision, square matrix
N	1	4×1	4-byte single precision, scalar
L	1	8×1	8-byte double precision, scalar
Δx	1	8×1	8-byte double precision, scalar
k	1	8×1	8-byte double precision, scalar
Total		$8N^2 + 16N + 28$	

TABLE 4. Memory requirement for 2D, fourth order central difference

Variable	Dimension	Memory	Remarks
T	N^2	$8 \times N^2$	8-byte double precision, column vector
B	N^2	$8 \times N^2$	8-byte double precision, column vector
A	$N^2 \times N^2$	$8 \times N^2 \times N^2$	8-byte double precision, square matrix
N	1	4×1	4-byte single precision, scalar
L	1	8×1	8-byte double precision, scalar
Δx	1	8×1	8-byte double precision, scalar
k	1	8×1	8-byte double precision, scalar
Total		$8N^4 + 16N^2 + 28$	

7. Description of all the files

Assuming that you are in the `proj1/src` subdirectory.

7.1. Description of C++ files.

- `main_code.cpp` - This modular main code file parses input file then performs defensive checks. Then it sends the inputs to the corresponding functions for assembling and solving the **AT=q** problem. Finally, it gives output based on the output mode and stores the solution in `sol.dat`.
- `assemble.cpp` - It assembles the **A** matrix based on the inputs. i.e. N, dimension and order.
- `assemble_RHS.cpp` - It assembles the **q** matrix based on inputs (N, dimension, order, xmin and xmax) and uses `MASA` library.
- `solver.cpp` - Based on the inputs, it solves **AT=q** using either Gauss-Seidel or Jacobi. It uses L_2 norm for convergence.

- `assemble_Texact.cpp` - It assembles the $\mathbf{T}_{exact}$ matrix using Method of Manufactured Solution (MMS) based on inputs and by using `MASA` library.
- `verification_debug.cpp` - In the debug mode, it prints the matrix $\mathbf{A}$, $\mathbf{q}$, $\mathbf{T}$ and $\mathbf{T}_{exact}$ in MATLAB-readable format using `GRVY` library. In verification mode, it prints out the L_2 norm of error in $\mathbf{T}$.
- `write_output_file.cpp` - This file writes output to a file `sol.dat` and the output filename can be changed in the input file. However, changing it will not work for the later shell scripts used in convergence study and the regression tests. Best is to keep it that.
- `global_timer.h` - This is a short header file for using a common timer provided in `GRVY` library, in all the functions.

7.2. **Description of shell scripts.**

- `convergence_testing.sh` - This runs all the tests required for the convergence study in one-go. It changes the `input_file.dat` with all the permutations of dimension, order, N and solvers and stores the outputs for different N with time elapsed and L_2 norm of error in `results/output_*`. It also saves the `sol.dat` file with corresponding name of test case as `ref_sol*`. Running this first requires code compilation as given in section 8.1.
- `job.sh` - This runs the `convergence_testing.sh` script on Stampede2 with SLURM job inputs. To run this enter the command `sbatch job.sh`.
- `../tests/rng.sh` - This runs the specialized regression tests for our program. It utilizes Bash Automated Testing System (`bats`). To run this we first have to run `../build_autotools.sh` in main directory, then update the PATH for `bats` library and run `../tests/rng.sh`.

We'll talk about running the above scripts in more detail wherever necessary. Shell scripts sometimes require the command `chmod 700 *.sh` to change the permissions in case they are not working.

7.3. **Description of other files.**

- `input_file.dat` - This text file contains all the inputs provided to the program if the program is not used for convergence testing and the executable `solve`$_h$*eatcondisdirectlyrunwithoutrunning*`convergence_testing.sh`*script.*/`results/`
Thispythonscriptrunstogivetheconvergenceplotsandthesurfaceplotsofthesolutions.

7.4. **Description of output files.**

- `sol.dat` - This file is an output of the main exectuable `solve_heatcond`. It contains the arrays of x (and y) locations and the arrays $\mathbf{T}$ and $\mathbf{T}_{exact}$ in degree Celsius.
- `/results/output_*.dat` - This is an output of `convergence_testing.sh` and `job.sh` scripts related with convergence analysis. It contains the N, L_2 norm of error in $\mathbf{T}$ and the runtime for each test.
- `/results/ref_sol_*.dat` - This file contains the results for regression testing and verification exercise.
- `/results/convergence*.png` - This file has the convergence plots for all the tests coming from postprocessing `plotting.py` python code ran manually in Spyder IDE.
- `plot*.png` - This file has the 3D surface plots for all the tests coming from postprocessing `plotting.py` python code ran manually in Spyder IDE
- `runtime*.png` - This file has the runtime plots for all cases coming from postprocessing `plotting.py` python code ran manually in Spyder IDE

8. Procedure for building, running the Codes & Tests

8.1. **Running the codes & tests.**

8.1.1. *Running the main code.* The main code is in `proj1/src` subdirectory, which also contains `configure.ac` and `Makefile.am` files. `Makefile.am` is present in the main directory `proj1` and all the subdirectories. First, building autotools using `autoreconf -f -i` and exporting the path to Dr. Karl Schulz's $WORK directory and finally `configure` with `GRVY` and `MASA` libraries. Next, using `make` and finally going into the directory using `cd src` and running the executable `solve_heatcond`.

```
$ ./build_autotools.sh
$ make
$ cd src
$ ./solve_heatcond
```

8.1.2. *Running the regression tests.* The regression tests are available in `proj1/test` subdirectory inside the file `rng.sh`. To run these tests, in the main directory, first export path to Dr. Karl Schulz's `bats` library and then build autotools using `autoreconf -f -i` and. Finally, run the `make check`. The commands are:

```
$ export PATH=/work/00161/karl/stampede2/public/bats/bin/:$PATH
$ ./build_autotools.sh
$ make check
```

To observe the individual regression tests we are supposed to run `rng.sh` inside `test` subdirectory:

```
$ export PATH=/work/00161/karl/stampede2/public/bats/bin/:$PATH
$ ./build_autotools.sh
$ cd test
$ ./rng.sh
```

8.1.3. *Running the post processing python script.* The python script is ran in Anaconda Spyder IDE, however, it can be run by different means.

8.2. **Input options.**

8.2.1. *Description of input file.* The `input_file.dat` data file contains two sections: `mesh` and `solver`. In the `mesh`, you can input the dimensions (1 or 2), grid (N), `xmin`(= `ymin`) and `xmax`(= `ymax`). For the solver, order of accuracy (2 or 4), type of solver (Jacobi or Gauss-Seidel), verification mode (1 for on or else off), output mode (0 for silent or 1 for debug), thermal conductivity k, tolerance for iterative solver (`eps`), maximum number of iterations for the iterative solvers (`max_iter`) and finally the name of the output data file `output_file`. An example of the same with debug mode on and verification mode on for Jacobi method, order 2, dimension 1 with $N = 5$ is given in Figure 3.

8.2.2. *Verification mode.* : The verification mode is switched on in the input file `input_file.dat` by `verify_mode=1`. It is off for every other input. The verification mode provides the L_2 norm of the error in temperatures T with respect to the exact temperatures provided by `MASA` library. An example of the output with debug mode off and verification mode on for Jacobi method, order 2, dimension 1 with $N = 5$ is given in Figure 4.

8.2.3. *Debug mode.* : The debug mode is switched on in the input file `input_file.dat` by `output_mode=1`. It is off for silent mode (=0). The debug output mode provides all the arrays $\mathbf{A}$, $\mathbf{T}$, $\mathbf{q}$, $\mathbf{T}_{exact}$ in MATLAB output form for debugging the code. An example of output with debug mode on and verification mode off for Jacobi method, order 2, dimension 1 with $N = 5$ is given in Figure 5.

mashadab@login1.stampede2:~/cse380-2020-student-mashadab/proj1/src

```
####Input file for heat conduction problem####

[mesh]

dimension = 1                   # 1 or 2 dimensions?
xmin = 0.0                      # min x location [m]
xmax = 1.0                      # max x location [m]
grid = 5                        ### number of mesh points in x-direction

[solver]

fd_method = 2                   # 2 = second order, 4 = fourth order
iter_method = Jacobi            # choose Jacobi or Gauss-Seidel
verify_mode = 1                 # enable verification mode with MASA = 1. disable = else
output_mode = 1                 # output mode (0=silent, 1=debug)
k = 1.0                         # thermal conductivity [W/mK]
eps = 1.0e-12                   # iterative solver tolerance
max_iter = 250000               # max solver iterations
output_file = 'sol.dat'         # name of solution output file
~
~
```

FIGURE 3. Example of inputfile with debug and verification modes on for Jacobi method, order 2, dimension 1 with $N = 5$

9. SOFTWARE TESTING

9.1. **Verification testing.** The code is working fine where we get the L_2 norm of error which reduces with a rise in grid cells N and we achieve the desired order of convergence. An example of the verification test is shown in Figure 6.

Order of convergence analysis:

(1) Jacobi: The solver did not converge for fourth order central differencing, so we will only show the results for second order finite differencing in both 1D and 2D. We achieved desired order of convergence as shown in Figure 7.
(2) Gauss-Seidel: Since Gauss-Seidel took long time to converge, we have gone to $N \times N = 75 \times 75$ in 2D but $N = 150$ in 1D. We have achieved desired order of convergence with Gauss-Seidel solver for both second and fourth order finite differencing, as shown in Figure 8.

9.2. **Regression testing.** The output of regression testing is given in Figure 9 for the `make check` and Figure 10 for the `./test/rng.sh`. We successfully passed the 5 tests that we designed, which are:

- Verifying the successful code compilation

```
mashadab@login1.stampede2:~/cse380-2020-student-mashadab/proj1/src
####Input file for heat conduction problem####
login1.stampede2(1145)$ vim input_file.dat
login1.stampede2(1146)$ vim input_file.dat
login1.stampede2(1147)$ login1.stampede2(1147)$ ./solve_heatcond
--> dimension   = 1
(fread: Using pre-registered value for variable): mesh/xmin -> 0
--> xmin        = 0.000000
(fread: Using pre-registered value for variable): mesh/xmax -> 1
--> xmax        = 1.000000
--> grid        = 5
--> fd_method   = 2
--> verify_mode = 1
(fread: Using pre-registered value for variable): solver/output_mode -> 0
--> output_mode = 0
--> k           = 1.000000
--> iter_method = Jacobi
--> eps         = 0.000000
--> max_iter    = 250000
--> output_file = sol.dat
--------> VERIFICATION MODE <--------
L2 Norm of error for n 5 is 0.1533

-------------------------------------------------------------------------------------------------------------
GRVY Performance timing - Performance Timings:                 |      Mean      Variance       Count
--> main                        : 3.36552e-03 secs ( 52.3726 %) | [3.36552e-03  0.00000e+00          1]
--> q_1D_order2                 : 1.77193e-03 secs ( 27.5739 %) | [1.77193e-03  0.00000e+00          1]
--> Texact_1D_order2            : 1.18303e-03 secs ( 18.4098 %) | [1.18303e-03  0.00000e+00          1]
--> L2_norm                     : 4.07696e-05 secs (  0.6344 %) | [2.66468e-07  2.19899e-13        153]
--> verification_mode           : 2.90871e-05 secs (  0.4526 %) | [2.90871e-05  0.00000e+00          1]
--> jacobi                      : 1.07288e-05 secs (  0.1670 %) | [1.07288e-05  0.00000e+00          1]
--> main_assemble_A             : 4.05312e-06 secs (  0.0631 %) | [4.05312e-06  0.00000e+00          1]
--> main_solver                 : 1.90735e-06 secs (  0.0297 %) | [1.90735e-06  0.00000e+00          1]
--> printing_matrix_A           : 9.53674e-07 secs (  0.0148 %) | [9.53674e-07  0.00000e+00          1]
--> printing_q_Tcomputed_Texact : 0.00000e+00 secs (  0.0000 %) | [0.00000e+00  0.00000e+00          1]
--> GRVY_Unassigned             : 1.52588e-05 secs (  0.2375 %)

           Total Measured Time = 6.42610e-03 secs ( 99.9555 %)
```

FIGURE 4. Example of output for verification mode on with Jacobi method, order 2, dimension 1 with $N = 5$.

- Verifying if verification mode runs properly
- Verifying if debug mode runs properly
- Verifying that the Gauss-Seidel solver matches with reference outputs
- Verifying that the Jacobi solver matches with reference outputs

9.3. **Runtime performance testing.** An example of run time results is given in Figure 11.

Runtime analysis:

(1) Jacobi: The run time increases with N, as expected. From Figure 12, it can be seen that time for 2D case is almost 3 orders of magnitude higher than for 1D case for same N as the number of grids in former is the square of the latter.

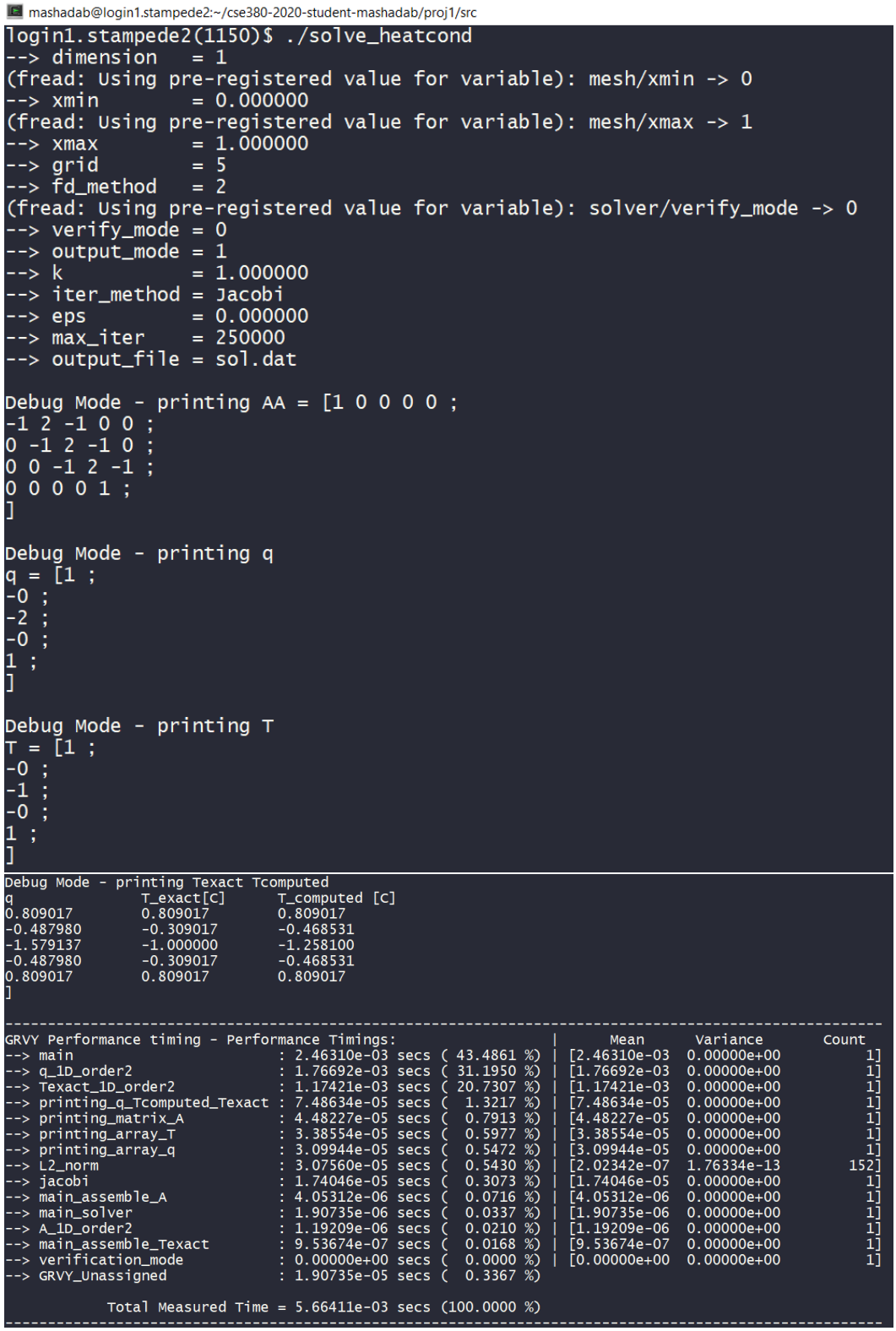


FIGURE 5. Example of output for verification mode on with Jacobi method, order 2, dimension 1 with $N = 5$.

```
mashadab@login1.stampede2:~/cse380-2020-student-mashadab/proj1/src
####Input file for heat conduction problem####
login1.stampede2(1145)$ vim input_file.dat
login1.stampede2(1146)$ vim input_file.dat
login1.stampede2(1147)$ login1.stampede2(1147)$ ./solve_heatcond
--> dimension   = 1
(fread: Using pre-registered value for variable): mesh/xmin -> 0
--> xmin        = 0.000000
(fread: Using pre-registered value for variable): mesh/xmax -> 1
--> xmax        = 1.000000
--> grid        = 5
--> fd_method   = 2
--> verify_mode = 1
(fread: Using pre-registered value for variable): solver/output_mode -> 0
--> output_mode = 0
--> k           = 1.000000
--> iter_method = Jacobi
--> eps         = 0.000000
--> max_iter    = 250000
--> output_file = sol.dat
--------> VERIFICATION MODE <--------
L2 Norm of error for n 5 is 0.1533

-------------------------------------------------------------------------------------------------------------
GRVY Performance timing - Performance Timings:                 |      Mean      Variance       Count
--> main                        : 3.36552e-03 secs ( 52.3726 %) | [3.36552e-03  0.00000e+00          1]
--> q_1D_order2                 : 1.77193e-03 secs ( 27.5739 %) | [1.77193e-03  0.00000e+00          1]
--> Texact_1D_order2            : 1.18303e-03 secs ( 18.4098 %) | [1.18303e-03  0.00000e+00          1]
--> L2_norm                     : 4.07696e-05 secs (  0.6344 %) | [2.66468e-07  2.19899e-13        153]
--> verification_mode           : 2.90871e-05 secs (  0.4526 %) | [2.90871e-05  0.00000e+00          1]
--> jacobi                      : 1.07288e-05 secs (  0.1670 %) | [1.07288e-05  0.00000e+00          1]
--> main_assemble_A             : 4.05312e-06 secs (  0.0631 %) | [4.05312e-06  0.00000e+00          1]
--> main_solver                 : 1.90735e-06 secs (  0.0297 %) | [1.90735e-06  0.00000e+00          1]
--> printing_matrix_A           : 9.53674e-07 secs (  0.0148 %) | [9.53674e-07  0.00000e+00          1]
--> printing_q_Tcomputed_Texact : 0.00000e+00 secs (  0.0000 %) | [0.00000e+00  0.00000e+00          1]
--> GRVY_Unassigned             : 1.52588e-05 secs (  0.2375 %)

           Total Measured Time = 6.42610e-03 secs ( 99.9555 %)
```

FIGURE 6. Example of output for verification mode on with Jacobi method, order 2, dimension 1 with $N = 5$.

(2) Gauss-Seidel: The run time increases with N, as expected. From Figure 13, it can be seen that time for 2D case is almost 2 orders of magnitude higher than for 1D case for same N as the number of grids in former is the square of the latter. Also, we can observe that the time for second and fourth order central differencing are the very similar, although the values are little higher for latter. But interestingly, it busts the myth that higher order methods are more expensive. By increasing the order, the errors are reducing by N^{-4} even though the computational cost is very similar, proving the utility of high order methods.

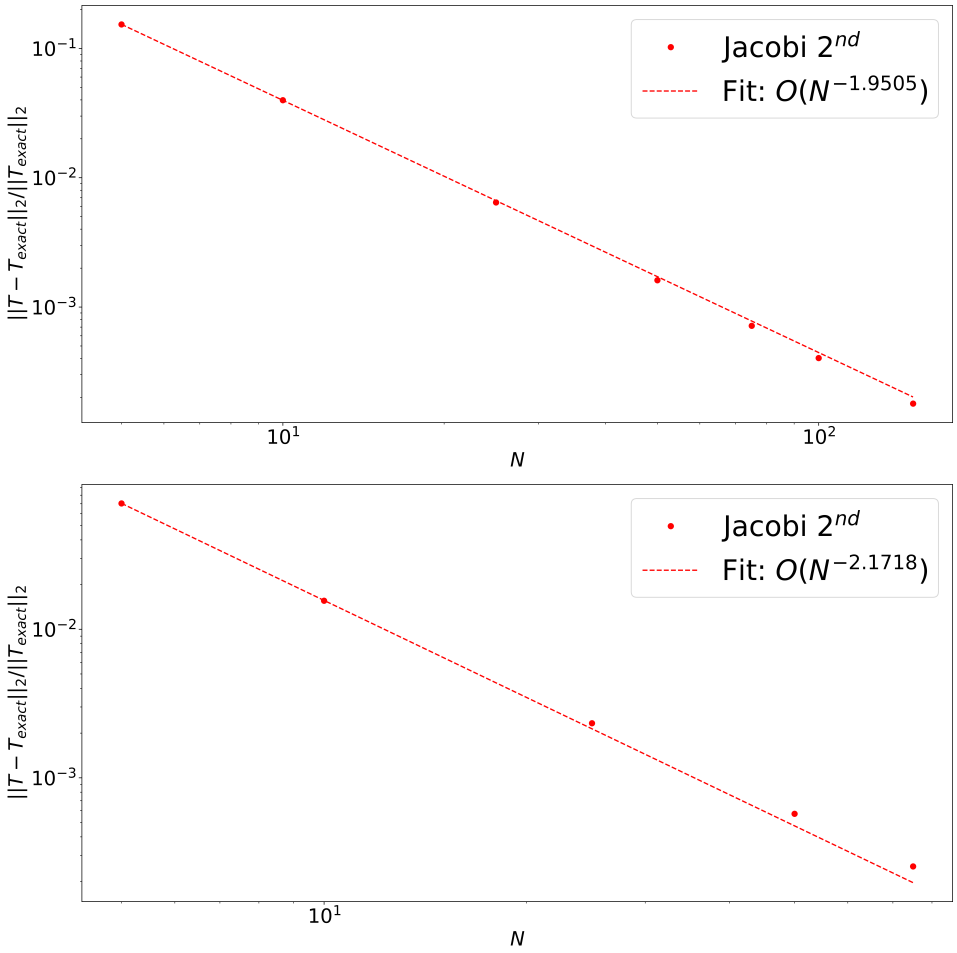


FIGURE 7. Convergence study for 1D (top) and 2D (bottom) cases for second order central differencing using Jacobi iterative solver.

10. RESULTS

The results for one dimensional tests are given in Figure 14 and for two-dimensional tests are given in Figure 15.

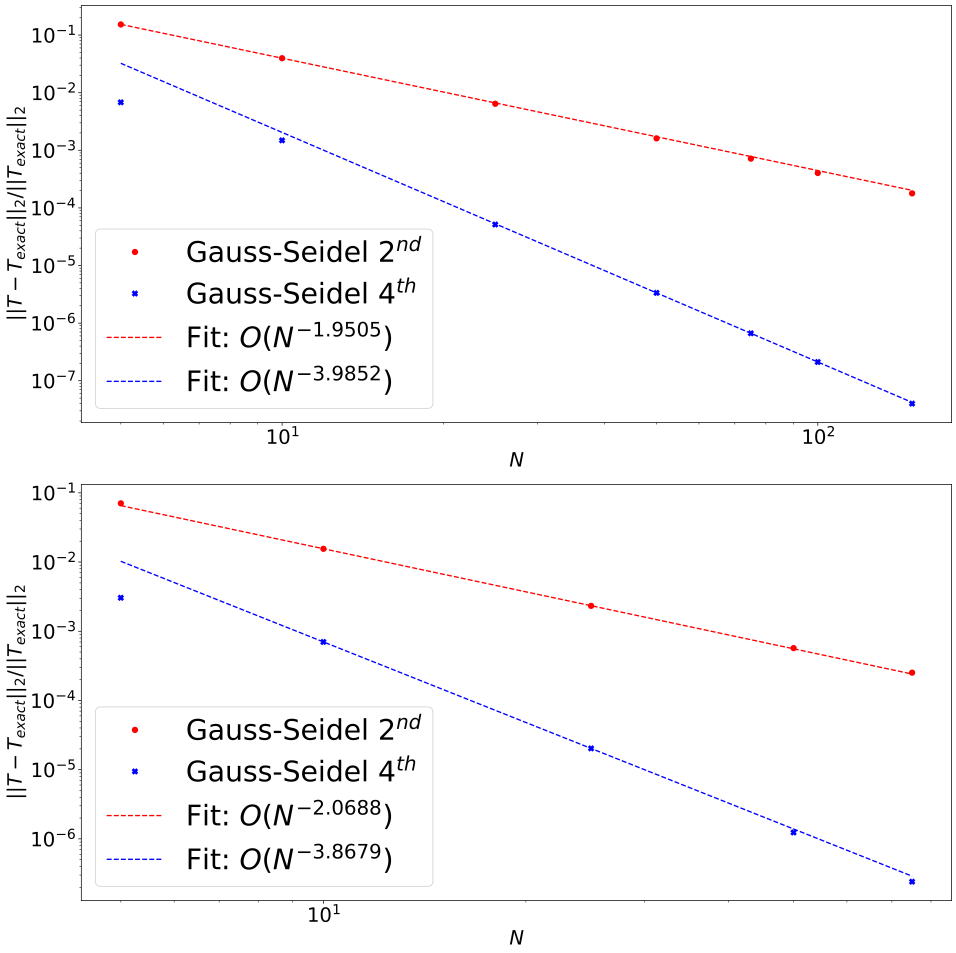


FIGURE 8. Convergence study for 1D (top) and 2D (bottom) cases for two and fourth order central differencing using Gauss-Seidel iterative solver.

REFERENCES

[1] Kendall E Atkinson and Weimin Han. *Elementary numerical analysis*. Wiley New York, 1985.

[2] J Tinsley Oden. *An introduction to mathematical modeling: a course in mechanics*. John Wiley & Sons, 2011.

[3] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

Email address: mashadab@utexas.edu

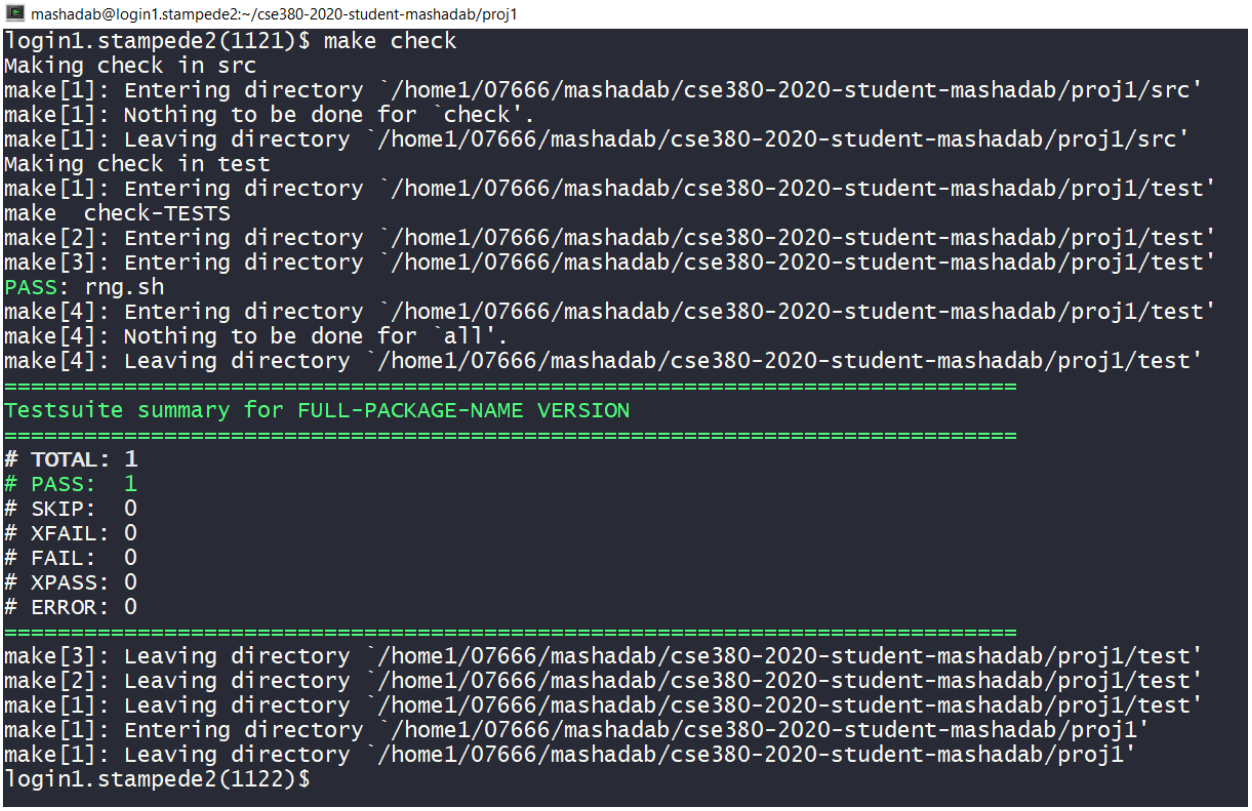


FIGURE 9. Regression testing by running `make check` in the main directory. We successfully passed the test `/test/rng.sh`, shown in green color.

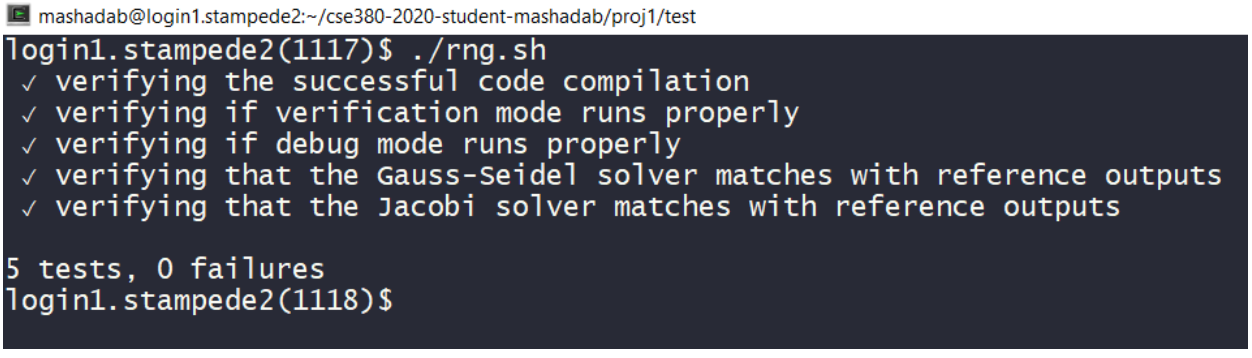


FIGURE 10. Regression testing by running the file `/test/rng.sh`.

```
mashadab@login1.stampede2:~/cse380-2020-student-mashadab/proj1/src
login1.stampede2(1153)$ ./solve_heatcond
--> dimension   = 1
(fread: Using pre-registered value for variable): mesh/xmin -> 0
--> xmin        = 0.000000
(fread: Using pre-registered value for variable): mesh/xmax -> 1
--> xmax        = 1.000000
--> grid        = 5
--> fd_method   = 2
(fread: Using pre-registered value for variable): solver/verify_mode -> 0
--> verify_mode = 0
(fread: Using pre-registered value for variable): solver/output_mode -> 0
--> output_mode = 0
--> k           = 1.000000
--> iter_method = Jacobi
--> eps         = 0.000000
--> max_iter    = 250000
--> output_file = sol.dat

-------------------------------------------------------------------------------------------------------------
GRVY Performance timing - Performance Timings:                  |      Mean      Variance       Count
--> main                        : 3.01623e-03 secs ( 49.7738 %) | [3.01623e-03  0.00000e+00          1]
--> q_1D_order2                 : 1.78003e-03 secs ( 29.3740 %) | [1.78003e-03  0.00000e+00          1]
--> Texact_1D_order2            : 1.18208e-03 secs ( 19.5066 %) | [1.18208e-03  0.00000e+00          1]
--> L2_norm                     : 4.19617e-05 secs (  0.6924 %) | [2.76064e-07  2.31940e-13        152]
--> jacobi                      : 1.40667e-05 secs (  0.2321 %) | [1.40667e-05  0.00000e+00          1]
--> main_assemble_A             : 4.05312e-06 secs (  0.0669 %) | [4.05312e-06  0.00000e+00          1]
--> main_solver                 : 1.90735e-06 secs (  0.0315 %) | [1.90735e-06  0.00000e+00          1]
--> A_1D_order2                 : 1.19209e-06 secs (  0.0197 %) | [1.19209e-06  0.00000e+00          1]
--> verification_mode           : 9.53674e-07 secs (  0.0157 %) | [9.53674e-07  0.00000e+00          1]
--> main_assemble_q             : 0.00000e+00 secs (  0.0000 %) | [0.00000e+00  0.00000e+00          1]
--> GRVY_Unassigned             : 1.26362e-05 secs (  0.2085 %)

            Total Measured Time = 6.05989e-03 secs ( 99.9213 %)

[*] Warning: Profile percentages do not sum to 100 %.
This likely means that you defined timer keys which are
not mutually exclusive.
-------------------------------------------------------------------------------------------------------------
```

FIGURE 11. Example of runtime performance output with Jacobi method, order 2, dimension 1 with $N = 5$.

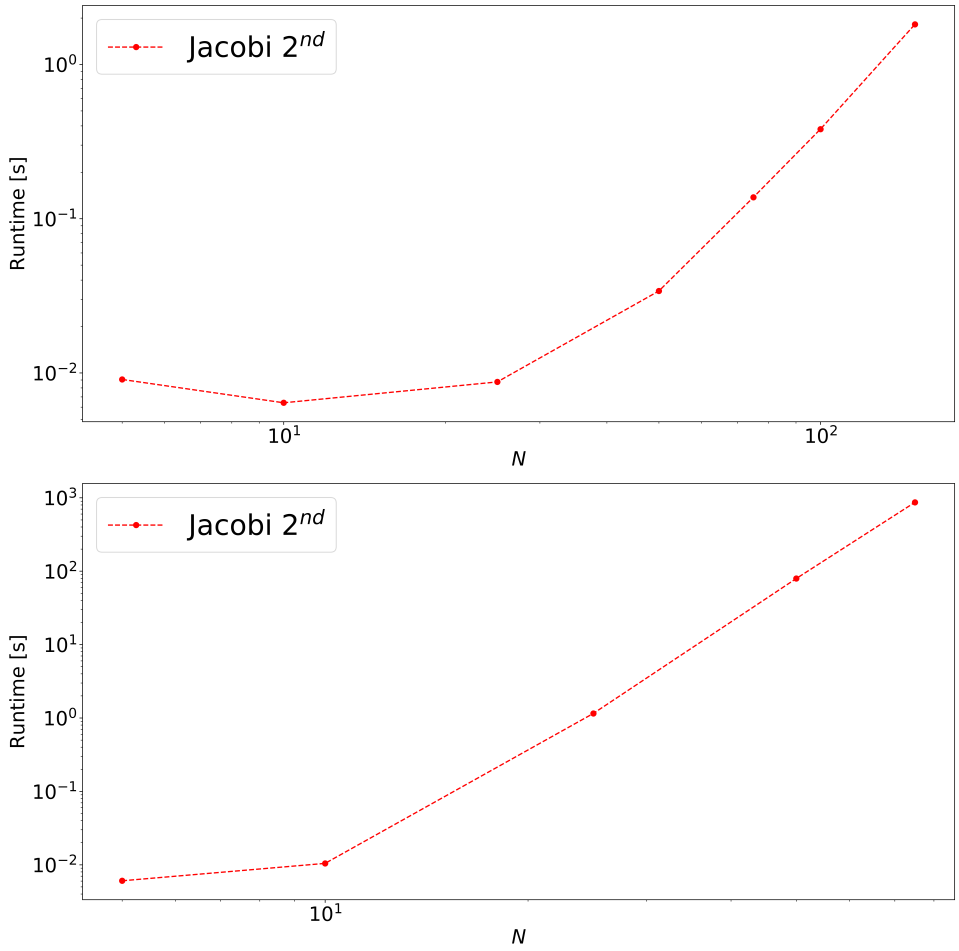


FIGURE 12. CPU Runtimes for 1D (top) and 2D (bottom) cases for second order central differencing using Jacobi iterative solver using 1 skx node and 48 cores on Stampede 2 supercomputer at TACC.

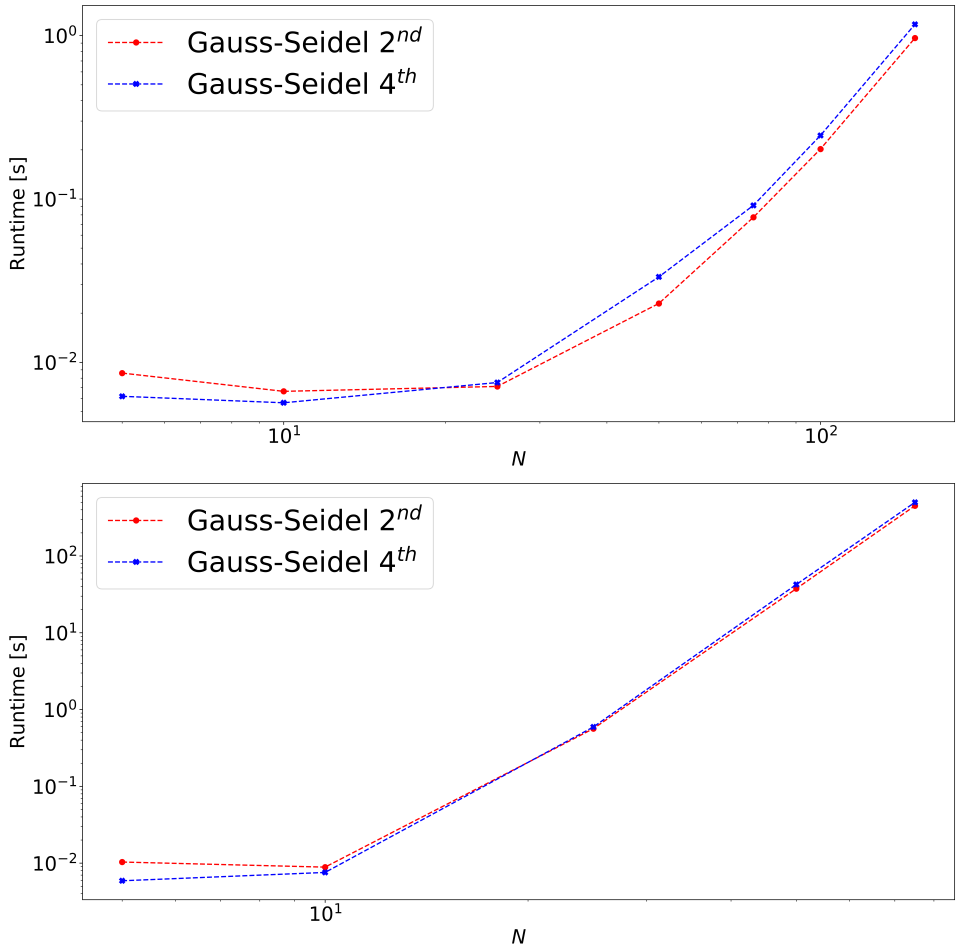


FIGURE 13. CPU Runtimes for 1D (top) and 2D (bottom) cases for second and fourth order central differencing using Gauss-Seidel iterative solver using 1 node and 48 cores using 1 skx node and 48 cores on Stampede 2 supercomputer at TACC.

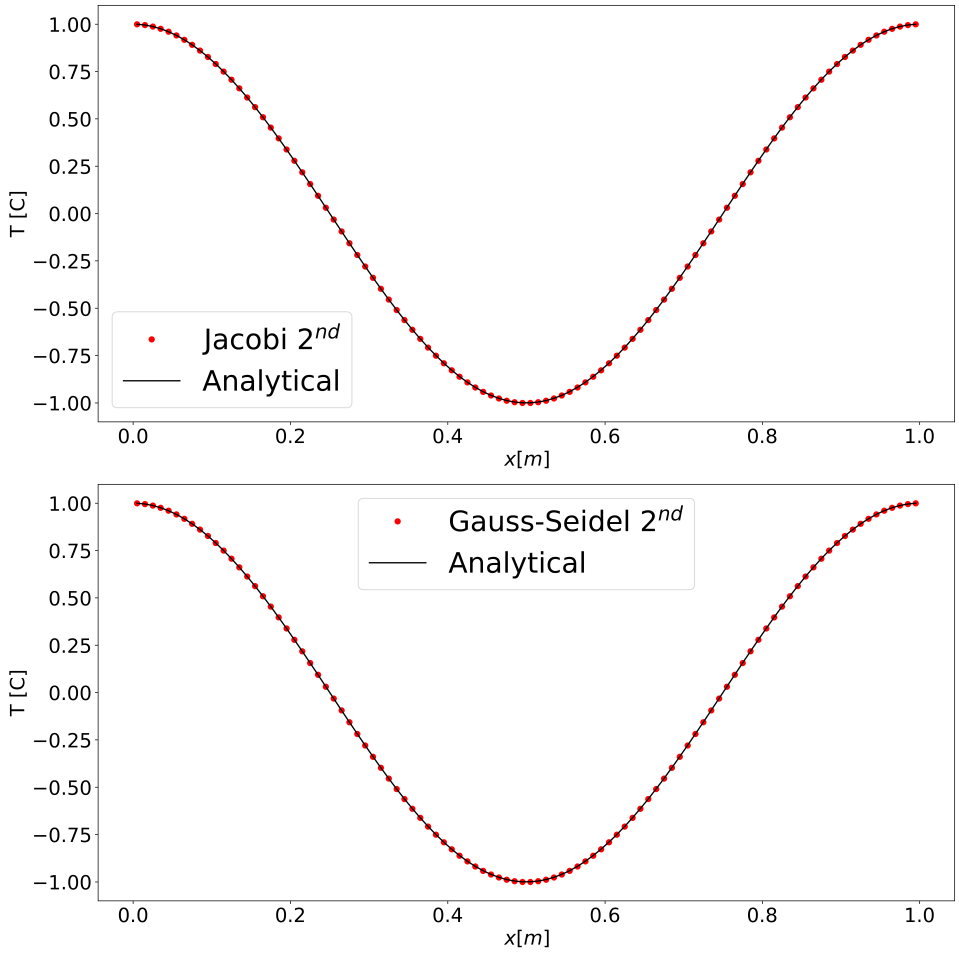


FIGURE 14. Results for the case of 1D tests, second order finite differencing, $N = 100$ using Jacobi (top) and Gauss-Seidel (bottom) iterative solvers.

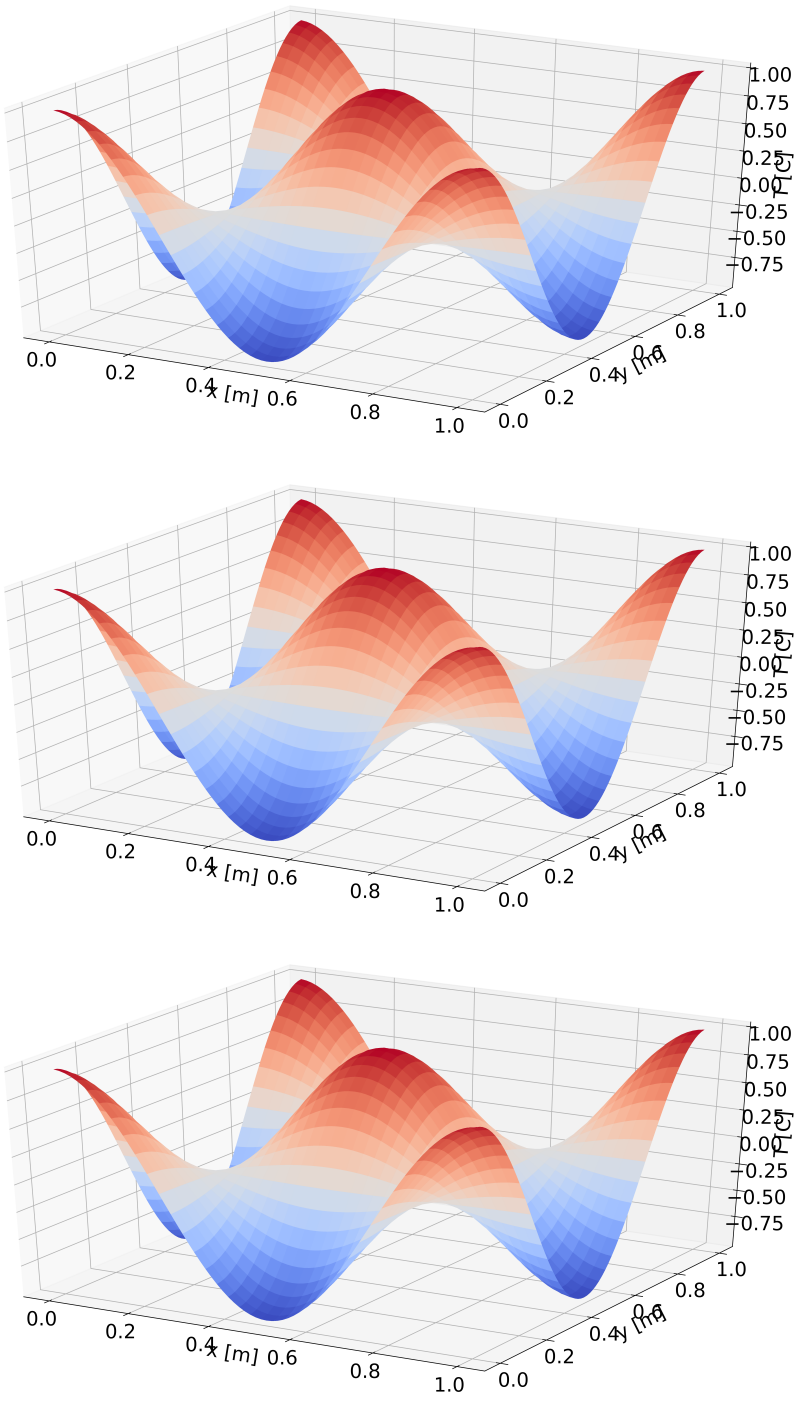


FIGURE 15. Results for the case of 2D tests, second order finite differencing, $N = 75 \times 75$ using Jacobi (top), Gauss-Seidel (middle) iterative solvers and the bottom one is the analytical result.