Numerical Methods in Steady State 1D and 2D Heat Equations

Yuege Xie (EID:yx4256)

Contents

1 Introduction

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

$$-k\nabla^2 u(x,y) = q(x,y)$$

where k is the thermal conductivity, u is the material temperature, and q is a heat source term. But in the code, I use f instead of q, it is just a notation.

In this Documentation, we first list some assumptions that help us to simplify and reformulate the 1D and 2D problems. Second, we derive the 2nd and 4th order finite difference expression using node-based central difference. Third, we transform the PDE into linear systems with certain matrices by flattening. Then, we can use Jacobi and Gauss-Seidel iterative methods to solve PDE numerically by solving linear systems.

2 Assumptions

- **Dirichlet Boundary Condition:** The solution is known on the boundary, i.e.

  $$u(x)|_{x=a,b} = u_0(x)$$
  $$u(x,y)|_{\partial\Omega} = u_0(x,y)$$

  for 1D and 2D cases, respectively, hence the scheme is node-base.
- **Central Finite Difference Method:** use central finite difference for both 2nd order and 4th order approximations.
- **Special Points in 4th Order:** We assume that we know the boundary points in the case also for $i = 1, N-1; j = 1, N-1$ as inputs.
- **Domain Size:** In 2D case, the domain is rectangular, i.e $\Omega = [a_1, b_1] \times [a_2, b_2]$. Moreover, if $b_1 - a_1 = b_2 - a_2$, it's square, we consider this case.
- **Mesh Size:** $\Delta x = \Delta y = h$, i.e. using square mesh.
- **Scheme:** The computational scheme is node-based.
- **Smoothness:** u is smooth enough that we can do the Taylor expansion.

3 Steady State Heat Equation Numerical Formulation

3.1 Governing Equations

3.1.1 1D Case

Using assumptions above, we can reformulate 1D steady-state heat equation with dirichlet boundary condition into:

$$-k\nabla^2 u(x) = q(x), \quad \forall x \in \Omega = [a,b] \tag{1}$$

$$u(a) = u_0(a), \quad u(b) = u_0(b) \tag{2}$$

3.1.2 2D Case

Using assumptions above, we can reformulate 2D steady-state heat equation with dirichlet boundary condition into:

$$-k\nabla^2 u(x,y) = q(x,y), \quad (x,y) \in \Omega = [a_1, b_1] \times [a_2, b_2] \tag{3}$$

$$u(x,y)|_{\partial\Omega} = u_0(x,y) \tag{4}$$

3.2 Generate Grid Points

For 1D case, assume we have $N+1$ points, and let $h = \Delta x$, then

$$x_i = a + i * h, i = 0, 1, 2, ..., N$$

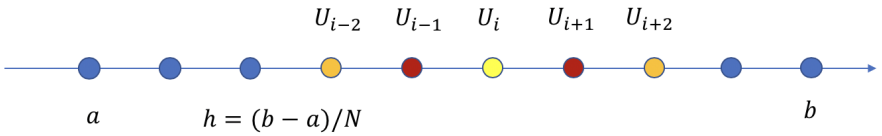


Figure 1: Node-based 1D discretized mesh

Let $\Delta x = \Delta y = h = \frac{b_i - a_i}{N_i}$ be the distance between two grid points. Indeed, for rectangular domain, we only have to choose different N_1 and N_2 to get the same h. Then

$$x_i = a_1 + i * h, i = 0, 1, 2, ..., N$$

$$y_j = a_2 + j * h, i = 0, 1, 2, ..., N$$

We want to get approximations of $U_{ij} = u(x_i, y_j)$ at grid points (x_i, y_j). Since by Dirichlet Boundary Condition, the solution on boundaries are known, we only have $(N-1) \times (N-1)$ unknowns to solve.

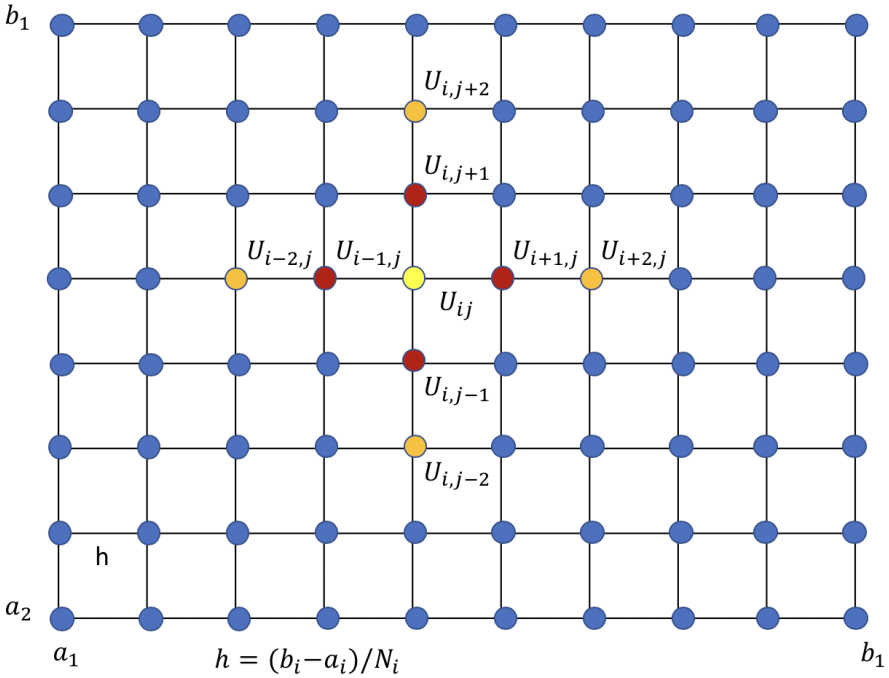


Figure 2: Node-based 2D discretized mesh

3.3 2nd-order Finite Difference Approximation

3.3.1 1D Case

Derived from Taylor expansion of u(x,y), we have

$$\begin{aligned}\nabla^2 u(x_i) =& \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{h^2} - \frac{2h^2}{4!}\frac{d^4u}{dx_i^4} + O(h^4) \\ =& \frac{U_{i-1} - 2U_i + U_{i+1}}{h^2} + T_i\end{aligned} \tag{5}$$

where the local truncation error $T_i = -\frac{2h^2}{4!}\frac{d^4u}{dx_i^4} + O(h^4)$ and we have $\lim_{h\to 0} T_i = 0, \forall i, j$, we can get the following equation:

$$\frac{-k}{h^2}[U_{i-1} - 2U_i + U_{i+1}] = q_i \tag{6}$$

where $q_i = q(x_i)$, $i = 1, 2, ..., N-1$.

3.3.2 2D Case

Derived from Taylor expansion of u(x,y), we have

$$\begin{aligned}\nabla^2 u(x_i, y_j) =& \frac{u(x_{i-1}, y_j) - 2u(x_i, y_j) + u(x_{i+1}, y_j)}{h^2} + \frac{u(x_i, y_{j-1}) - 2u(x_i, y_j) + u(x_i, y_{j+1})}{h^2} \\ & - \frac{2h^2}{4!}(\frac{\partial^4 u}{\partial x_i^4} + \frac{\partial^4 u}{\partial y_j^4}) + O(h^4) \\ =& \frac{U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4U_{ij}}{h^2} + T_{ij}\end{aligned} \tag{7}$$

where the local truncation error $T_{ij} = -\frac{2h^2}{4!}(\frac{\partial^4 u}{\partial x_i^4} + \frac{\partial^4 u}{\partial y_j^4}) + O(h^4)$ and we have $\lim_{h\to 0} T_{ij} = 0, \forall i, j$, we can get the following equation:

$$\frac{-k}{h^2}[U_{i,j-1} + U_{i-1,j} - 4U_{ij} + U_{i+1,j} + U_{i,j+1}] = q_{ij} \tag{8}$$

where $i = 1, 2, ..., N-1, j = 1, 2, ..., N-1$ and $q_{ij} = q(x_i, y_j)$

3.4 4th-order Finite Difference Approximation

3.4.1 1D Case

$$\nabla^2 u(x_i) = \frac{-U_{i-2,j} + 16U_{i-1,j} - 30U_{ij} + 16U_{i+1,j} - U_{i+2,j}}{12h^2} - \frac{8h^4}{6!}\frac{d^6u}{dx_i^6} + O(h^6) \tag{9}$$

where $\lim_{h\to 0} -\frac{8h^4}{6!}\frac{d^6u}{dx_i^6} + O(h^6) = 0, \forall i$. Then we get:

$$\frac{-k}{h^2}[-\frac{1}{12}U_{i-2} + \frac{4}{3}U_{i-1} - \frac{5}{2}U_i + \frac{4}{3}U_{i+1} - \frac{1}{12}U_{i+2}] = q_i \tag{10}$$

where $i = 2, ..., N-2$ and $q_i = q(x_i)$.

3.4.2 2D Case

Similar as above, ignore the high order term of h, we have

$$\begin{aligned}\nabla^2 u(x_i, y_j) =& \frac{-U_{i-2,j} + 16U_{i-1,j} - 30U_{ij} + 16U_{i+1,j} - U_{i+2,j}}{12(\Delta x)^2} \\ &+ \frac{-U_{i,j-2} + 16U_{i,j-1} - 30U_{i,j} + 16U_{i,j+1} - U_{i,j+2}}{12(\Delta y)^2} + T_{ij}\end{aligned} \tag{11}$$

where the local truncation error $T_{ij} = -\frac{8h^4}{6!}(\frac{\partial^6 u}{\partial x^6} + \frac{\partial^6 u}{\partial y^6}) + O(h^6)$ $\lim_{h\to 0} T_{ij} = 0, \forall i, j$ as well, we can get the equations:

$$\frac{-k}{h^2}[-\frac{1}{12}U_{i-2,j} + \frac{4}{3}U_{i-1,j} - 5U_{ij} + \frac{4}{3}U_{i+1,j} - \frac{1}{12}U_{i+2,j} - \frac{1}{12}U_{i,j-2} + \frac{4}{3}U_{i,j-1} + \frac{4}{3}U_{i,j+1} - \frac{1}{12}U_{i,j+2}] = q_{ij} \tag{12}$$

where $i = 2, ..., N-2, j = 2, ..., N-2$ and $q_{ij} = q(x_i, y_j)$.

3.5 Linear System of Heat Equation and Matrix Form

3.5.1 1D Case

For 1D case, just let $z = [U_0, U_1, ..., U_{N-1}, U_N]^T$, where $U_0 = u_0(a), U_N = u_0(b)$, then we can get the $Az = b$ form, which can be solved by iterative methods. For 2nd order approximation, with Dirichlet boundary conditions known, we have the following form:

$$b_{2nd}^{1D} = [U_0, q_1, ..., q_{N-1}, U_N]^T$$

$$A_{2nd}^{1D} = \frac{-k}{h^2}\begin{bmatrix} \frac{-h^2}{k} & & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & & \frac{-h^2}{k} \end{bmatrix}$$

#nonzeros of interior: 3, except first and last is 1.
Note: Since for boundary points, we only need 1 for it, to make the matrix simple, I add $\frac{-h^2}{k}$ to recover it to 1.

For 4th order approximation,

$$b_{4th}^{1D} = [U_0, U_1, q_2, ..., q_{N-2}, U_N - 1, U_N]^T$$

$$A_{4th}^{1D} = \frac{-k}{h^2}\begin{bmatrix} \frac{-h^2}{k} & & & & & & \\ & \frac{-h^2}{k} & & & & & \\ -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & -\frac{1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & -\frac{1}{12} \\ & & & & & \frac{-h^2}{k} & \\ & & & & & & \frac{-h^2}{k} \end{bmatrix}$$

#nonzeros of interior row: 5, except first, second, first last and second last is 1.

3.5.2 2D Case

We want to transform the 2D problem into a linear system $Az = b$ as well and use iterative methods like Jacobi and Gauss-Seidel to solve it. First, assume $N_1 = N_2 = N$, flatten U_{ij} into a vector z using rule:

$$z_{i(N+1)+j} = U_{i,j}, \forall i = 0, 1, 2, ...N, j = 0, 1, 2, ..., N$$

Then we have $z = [U_{00}, U_{01}, ..., U_{0N}, U_{10}, ..., U_{NN}]^T$.
Second, we use the same rule of adding boundary variables as 1D case to flatten f into b:

$$b_{2nd}^{2D} = [U_{00}, U_{01}, ..., U_{0,N}, U_{10}, q_{11}, ..., q_{1,N-1}, ..., q_{N-1,N-1}, U_{N-1,N}, ..., U_{N,N}]^T$$

Third, we can get the form of corresponding A:

$$A_{2nd}^{2D} = \begin{bmatrix} I & & & & & & \\ I_{2nd} & B_{2nd} & I_{2nd} & & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & I_{2nd} & B_{2nd} & I_{2nd} & \\ & & & & & I & \end{bmatrix}$$

where I_{2nd} and B_{2nd} are $N + 1 \times N + 1$ matrix.
#nonzeros of interior row: 5, except special cases with 1.

$$B_{2nd} = \frac{-k}{h^2} \begin{bmatrix} \frac{-h^2}{k} & & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ & & & & \frac{-h^2}{k} \end{bmatrix}$$

$$I_{2nd} = \frac{-k}{h^2} \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \\ & & & & 0 \end{bmatrix}$$

For 4th order approximation, we know more boundary conditions including $i = 0, 1, N-1, N; j = 0, 1, N-1, N$, we have:

$$b_{4th}^{2D} = [U_{00}, U_{01}, ..., U_{0N}, U_{10}, U_{11}, ..., U_{1N}, U_{20}, q_{21}, ..., q_{N-2,N-2}, U_{N-1,N}, ..., U_{N,N}]^T$$

$$A_{4th}^{2D} = \begin{bmatrix} I & & & & & & & \\ & I & & & & & & \\ -\frac{1}{12}I_{4th} & \frac{4}{3}I_{4th} & B_{4th} & \frac{4}{3}I_{4th} & -\frac{1}{12}I_{4th} & & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & & \\ & & -\frac{1}{12}I_{4th} & \frac{4}{3}I_{4th} & B_{4th} & \frac{4}{3}I_{4th} & -\frac{1}{12}I_{4th} & \\ & & & & & I & & \\ & & & & & & I & \end{bmatrix}$$

where $I = I_{N+1 \times N+1}$ and B is also $N + 1 \times N + 1$ matrix.
#nonzeros of interior row: 9, except boundary cases are 1.

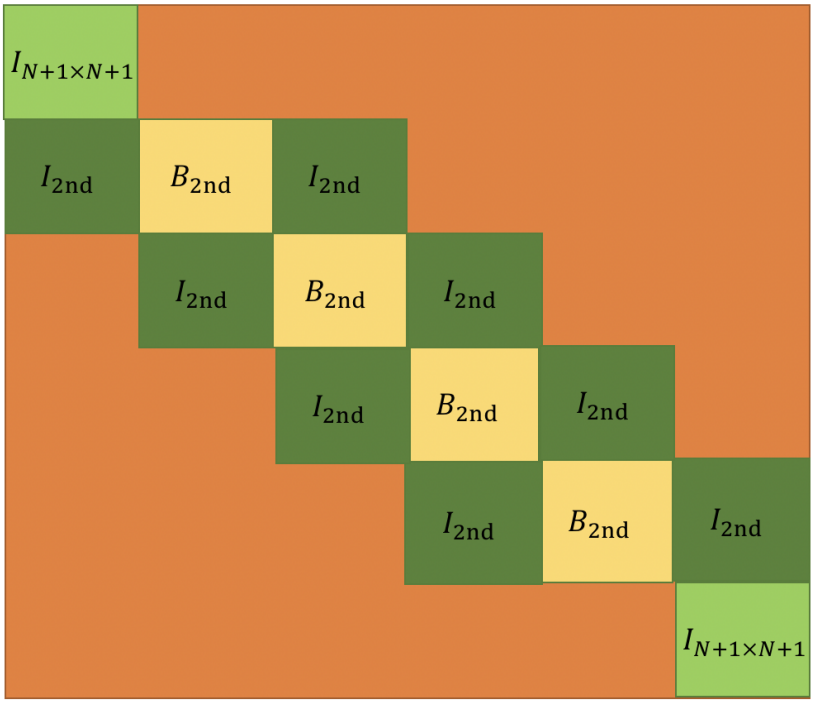


Figure 3: Structure of 2nd order difference matrix

$$B_{4th} = \frac{-k}{h^2}\begin{bmatrix} \frac{-h^2}{k} & & & & & & \\ & \frac{-h^2}{k} & & & & & \\ -\frac{1}{12} & \frac{4}{3} & -5 & \frac{4}{3} & -\frac{1}{12} & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & -\frac{1}{12} & \frac{4}{3} & -5 & \frac{4}{3} & -\frac{1}{12} \\ & & & & & \frac{-h^2}{k} & \\ & & & & & & \frac{-h^2}{k} \end{bmatrix}$$

$$I_{4th} = \frac{-k}{h^2}\begin{bmatrix} 0 & & & & & & \\ & 0 & & & & & \\ & & 1 & & & & \\ & & & \ddots & & & \\ & & & & 1 & & \\ & & & & & 0 & \\ & & & & & & 0 \end{bmatrix}$$

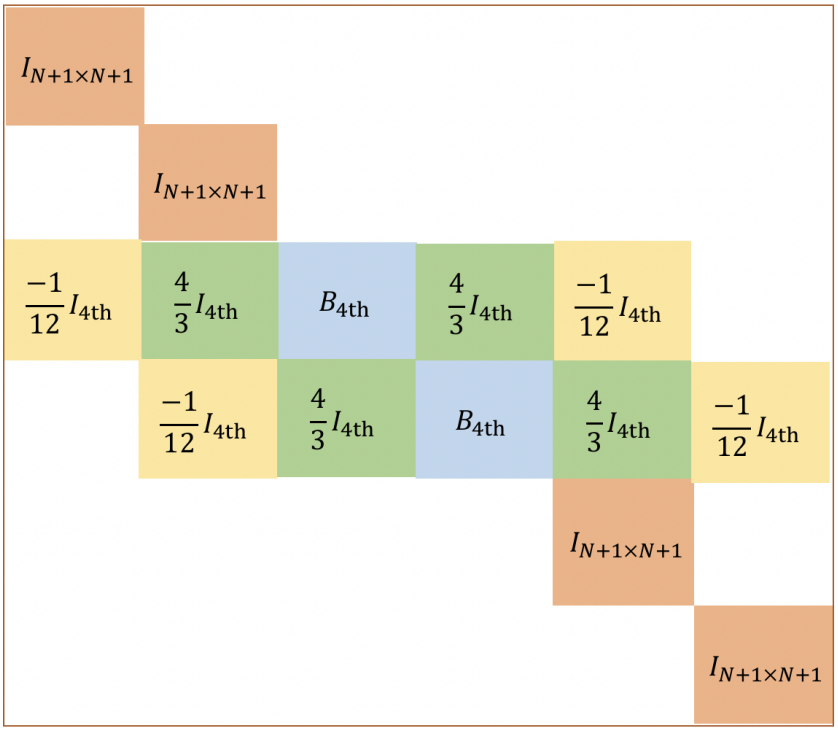


Figure 4: Structure of 4th order difference matrix

3.6 Iterative Methods to Solve Linear Systems

3.6.1 Jacobi Iterative Method

Given an initial guess $\mathbf{z}^0$, here we use 0 for initialization, then the rule of Jacobi iterative method is

$$z_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} z_j^k) \tag{13}$$

3.6.2 Gauss-Seidel Iterative Method

Given an initial guess $\mathbf{z}^0$, here we use 0 for initialization,then the rule of Gauss-Seidel iterative method is

$$z_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij} z_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} z_j^k) \tag{14}$$

Gauss-Seidel method use the newest to solve linear system, while Jacobi use the old one. For performance, Gauss-Seidel usually converges and use less memory, but not for Jacobi. In practice, we can see from the results, Jacobi does not converge for 4th order finite difference.

4 Algorithm

5 Memory Estimate

Memory that we need includes the following things:

Algorithm 1 Numerical Methods to Solve Steady State Heat Equations

Input: tolerance $\epsilon > 0$, max_iter, iter_method, dim, fd_method, domain, N, dim, output_mode, verification_mode, output_file
Output: $u(x_i, y_j), i = 1, 2, ..., N-1, j = 1, 2, ..., N-1$
Initialize: k = 0, error = 1000, $f(x_i, y_j)$
if dim == 1 **then**

$$z = [U_0, U_1, ..., U_{N-1}, U_N]^T$$

  if fd_method == 2nd **then**

$$b = b_{2nd}^{1D}$$
$$A = A_{2nd}^{1D}$$

  else

$$b = b_{4th}^{1D}$$
$$A = A_{4th}^{1D}$$

if dim == 2 **then**

$$z = [U_{00}, U_{01}, ..., U_{0N}, U_{10}, ..., U_{NN}]^T$$

  if fd_method == 2nd **then**

$$b = b_{2nd}^{2D}$$
$$A = A_{2nd}^{2D}$$

  else

$$b = b_{2nd}^{2D}$$
$$A = A_{4th}^{2D}$$

while error $> \epsilon$ and k <= max_iter **do**
  if iter_method == Jacobi **then**
    for $i = 0, 1, 2, .., N$ **do** sum = 0
      for $j = 0, 1, 2, ..., N$ **do**
        if $j \neq i$ **then** $sum = sum + a_{ij} z_j^k$
      $z_i^{k+1} = \frac{1}{a_{ii}}(b_i - sum)$
  if iter_method == Gauss_Seidel **then**
    for $i = 0, 1, 2, .., N$ **do** sum = 0
      for $j = 0, 1, 2, ..., i-1$ **do** $sum = sum + a_{ij} z_j^{k+1}$
      for $j = i+1, i+2, ..., N$ **do** $sum = sum + a_{ij} z_j^k$
      $z_i^{k+1} = \frac{1}{a_{ii}}(b_i - sum)$
  if iter_method == GMRES **then** use PETSC to get solution
  error = $\|z^{k+1} - z^k\|_2$
  $k \leftarrow k + 1$
return z^k

- **Iteration Variables:** matrix A, vector b, vector z, k, error
- **Inputs:** tolerance ϵ, max_iter, iter_method, dim, fd_method, domain, N, output_mode, verification_mode, output_file

1. For matrix A, because it has special diagonal structures, we can use sparse structure to store it. Use 1D array to save number of non-zeros in each row, let $n = N + 1$ for 1D case and $n = (N + 1) * (N + 1)$ for 2D case. Then we need n for non-zeros. Use 2D array to save non-zero column index and values, we need $3n$ for 1D 2nd, $5n$ for 1D 4th and 2D 2nd, $9n$ for 2D 4th.
2. For inputs and iteration variables k, error, we only need 1 double to store them, we can estimate 20 doubles for them.
3. For z and b, we need $N + 1$ for 1D case and $(N + 1)^2$ for 2D case, respectively.
4. For verification mode, we need n for exact value.

The difference between Jacobi and Gauss-Seidel iterations is that we update the variable in Gauss-Seidel iteration in the loop, so we only need one z to store it. But for Jacobi, we update them until we get all the new values for a new iteration, so we need at least two z to store them. But we need to compare the $l2$ norm for current and previous solution, so here we need one more n doubles for "old_z".

And we need 8B to store a double, then the total memory we need is as follows in Table 1. And we allocate dynamic memory to these variables.

iter_method/dim	1D	2D
2nd order	$(11 \times (N + 1) + 20) \times 8B$	$(15 \times (N + 1)^2 + 20) \times 8B$
4th order	$(15 \times (N + 1) + 20) \times 8B$	$(23 \times (N + 1)^2 + 20) \times 8B$

Table 1: Memory Estimate Summary

6 User Instructions

6.1 Build Procedures

Please follow the procedures to build heat equation solving systems on Stampede2,and the commands are also include in **"build_coverage.sh"** and **"build_petsc.sh"**, you can use "cat build_coverage.sh" and copy the commands for convenience.
Since the two can not be build at the same time, I strongly recommend you to build coverage at first, and use "make coverage" to get the coverage results, then build with PETSC to see other related results.

To enable coverage option, you can build as follows:

1. Download the tar of codes, "proj02" from Github, untar the files;
2. Use "autoreconf -f -i" to do the bootstrap;
3. Export PATHs;

```
export PKGPATH=/work/00161/karl/stampede2/public
export CLASSPATH=/work/00161/karl/stampede2/public
export MODULEPATH=$CLASSPATH/ohpc/pub/modulefiles/:$MODULEPATH
```

4. Load fixed toolchain, and use "which gcc" to check if the version is "/work/00161/karl/stampede2/public/ohpc/pub/

```
module swap intel gnu7
```

5. Load hdf5 module;

```
module load hdf5
```

6. Use the following command to do the configuration;

```
./configure CC=gcc --with-masa=$PKGPATH/masa-gnu7-0.50 \
--with-grvy=$PKGPATH/grvy-gnu7-0.34 \
--with-hdf5=$TACC_HDF5_DIR --enable-coverage
```

7. Use "make" to automatically build the system and "make check" to check it runs correctly;
8. Use "make coverage" to see the percentage of lines used in check process;
9. Use "cd src" to enter in the src file and "./solver input.dat" to solve heat equation, where you can change input.dat following the instructions in the file, which will also be illustrated in the "Input Options".

To enable PETSC option, you can build as follows:

1. Follow steps 1,2,3 the same as above and if you have built as above, remember to "make clean";
2. Load hdf5 and petsc modules;

```
module load hdf5
module load petsc
```

3. Use the following command to do the configuration;

```
./configure CC=mpicc --with-masa=$PKGPATH/masa-gnu7-0.50 \
--with-grvy=$PKGPATH/grvy-gnu7-0.34 \
--with-hdf5=$TACC_HDF5_DIR --with-petsc=$PETSC_DIR
```

4. Follow step 7 above to "make" and "make check";

5. Use step 9 to run, but you should use mpirun on computational node as follows:

```
srun --pty -N 1 -n 48 -t 15:00 -p skx-dev /bin/bash -l
mpirun -np 1 ./solver input.dat
```

6.2 Input Options

Input options can be changed in input.dat, which shows in Figure 5. Note that you can also change the name and the file name on the command line.

- k: thermal conductivity, which will be used as parameter for MASA
- verify_mode: 1 will enable verification mode with MASA and output error norm
- output_ mode: 0 = silent, 1 = standard, 2 = debug
- output_file: name of solution output file, for "make check", please set it as "sol.dat"
- dimensions: choose 1 for 1D and 2 for 2D
- xmin, xmax, ymin, ymax: set range of each axis
- N: number of intervals in one axis, points are $N + 1$ for 1D, $(N + 1)^2$ for 2D.
- fd_method: 2 = second order, 4 = fourth order
- iter_method: choose 1 for Jacobi or 2 for Gauss-Seidel or 3 for GMRES using PETSC
- eps: iterative solver tolerance, default is $1.0e - 12$
- max_iter: max solver iterations

```
# -*-sh-*-

# input file for solving heat conduct equation

k           = 1.0               # thermal conductivity [W/mK]
verify_mode = 1                 # enable verification mode with MASA
output_mode = 2                 # output mode (0 = silent, 1 = standard, 2 = debug)
output_file = 'sol.dat'         # name of solution output file


[mesh]

dimensions  = 1     # 1 or 2 dimensions?
xmin = 0            # min x location [m]
xmax = 1            # max x location [m]
ymin = 0            # min y location [m]
ymax = 1            # max y location [m]
N    = 128          # number of intervals in one axis, points are N+1 for 1D, (N+1)^2 for 2D.


[solver]

fd_method   = 2         # 2 = second order, 4 = fourth order
iter_method = 2         # choose 1 for Jacobi or 2 for Gauss-Seidel or 3 for GMRES
eps         = 1.0e-12   # iterative solver tolerance
max_iter    = 250000    # max solver iterations
```

Figure 5: Input options example

7 Code Coverage

We can use "make coverage" without PETSC and get the coverage results in the directory **"/coverage/lcov"** and Figure 6. You can check the "*.html" to see overall and detailed information. As we can see from the figure, the regression tests have 95.9% code coverage, which is bigger than 75%. And you can also see on the screen.

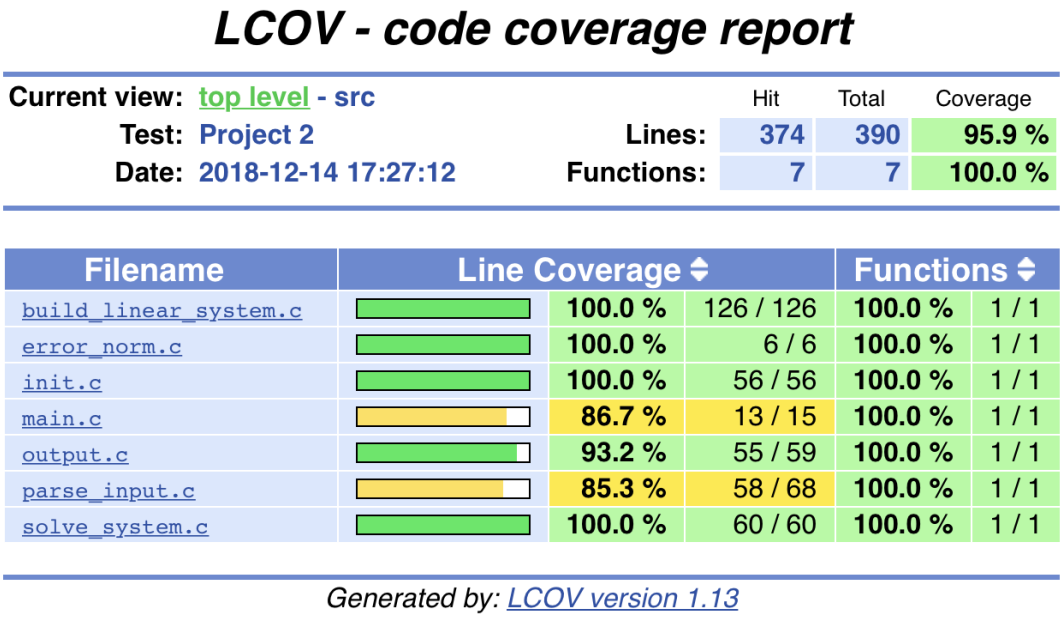
```
"Overall coverage rate:
  lines......: 95.9% (374 of 390 lines)
  functions..: 100.0% (7 of 7 functions)"
```

8 Verification Procedures and Exercise

Just change the input.dat with verify_mode to run in verification mode, since the silent output mode will show nothing, please change the output_mode to 1 or 2 to see the l_2 norm for the difference between the solution and the reference result generated from MASA. You can find example output for standard output in Figure 8, for debug output in Figure 9.

For verification exercise, we can run with $N = 8, 16, 32, 64, 128, 256$, and use "loglog" to plot and to illustrate the convergence rate with reference line. The expected slope for 2nd order is -2 and for 4th order is -4. We can see the result of Gauss-Seidel iterative method for 1D and 2D in Figure 10 and Figure 11, respectively; and Jacobi iterative method for 1D and 2D with 2nd order in Figure 12. They are all almost parallel to the reference line respectively.

From the results we can get slope as follows, Gauss and Jacobi are close in 2nd order.



LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - src				
Test:	Project 2	**Lines:**	374	390	95.9 %
Date:	2018-12-14 17:27:12	**Functions:**	7	7	100.0 %

Filename	Line Coverage			Functions	
build_linear_system.c		100.0 %	126 / 126	100.0 %	1 / 1
error_norm.c		100.0 %	6 / 6	100.0 %	1 / 1
init.c		100.0 %	56 / 56	100.0 %	1 / 1
main.c		86.7 %	13 / 15	100.0 %	1 / 1
output.c		93.2 %	55 / 59	100.0 %	1 / 1
parse_input.c		85.3 %	58 / 68	100.0 %	1 / 1
solve_system.c		100.0 %	60 / 60	100.0 %	1 / 1

Generated by: LCOV version 1.13

Figure 6: Code Coverage Demonstration

- 1d 2nd gauss: -2.0238
- 1d 4th gauss: -3.8642
- 2d 2nd gauss: -1.9933
- 2d 4th gauss: -3.8663
- 1d 2nd jacobi: -2.0038
- 2d 2nd jacobi: -1.9933

9 Runtime Performance

We use "GRVY" to get the summary of application time. As we can see from Figure 13, the 3 sections accounting for at least 90% are: parse_input, init and output, since the number of points are very small and solving process is very fast, the time spends more on read and output files. From Figure 14, the top three are solver_system parse_input and output, as the number of points rises, more time will spend on solver system procedure. When the number of points arrives at 256, over 99% time spends on solving procedure.

N / process %	solve_system	parse_input	output
8	0.91	46.87	25.16
32	71.54	11.29	9.19
256	99.88	0.0078	0.0667

Table 2: Runtime Summary

```
** Finite-difference based Heat Equation Solver (steady-state)
   --> Parsing runtime options from input.dat
   --> h          = 0.062500


** Runtime mesh settings  (1D):
   --> nx         = 16  (xmin,xmax) = (0.000000, 1.000000)


** Runtime solver settings:
   --> finite difference method = CENTRAL2
   --> iterative method = GAUSS_SEIDEL
   --> max iterations = 250000
   --> thermal conductivity = 1.000000
   --> solution output file = sol.dat


** Initializing data structures...


** Building linear system...
   --> Enforcing analytic Dirichlet BCs using MASA (1D)


** Solving linear system...
   --> Converged at iter: 526
   --> Writing output to sol.dat




** Computing l2 error norm.
   --> l2 error norm = 2.460876e-02


-----------------------------------------------------------------------------------------------
Steady Heat Equation Solver - Performance Timings:      |      Mean      Variance       Count
--> output              : 2.65288e-03 secs ( 36.2762 %) | [2.65288e-03  0.00000e+00          1]
--> parse_input         : 2.60901e-03 secs ( 35.6763 %) | [2.60901e-03  0.00000e+00          1]
--> init                : 1.92595e-03 secs ( 26.3359 %) | [1.92595e-03  0.00000e+00          1]
--> solve_system        : 9.29832e-05 secs (  1.2715 %) | [9.29832e-05  0.00000e+00          1]
--> build_linear_system : 2.00272e-05 secs (  0.2739 %) | [2.00272e-05  0.00000e+00          1]
--> GRVY_Unassigned     : 1.21593e-05 secs (  0.1663 %)

    Total Measured Time = 7.31301e-03 secs (100.0000 %)
-----------------------------------------------------------------------------------------------
```

Figure 7: Example of standard output for verification

```
** Solving linear system...
   --> Converged at iter: 92
   --> Writing output to sol.dat




** Computing l2 error norm.
   --> l2 error norm = 6.948365e-03
```

Figure 8: Standard for verification

```
[debug]: solve_system           - function end
[debug]: output                 - function begin
   --> Writing output to sol.dat
[debug]: output                 - function end
[debug]: error_norm             - function begin


** Computing l2 error norm.
   --> l2 error norm = 6.948365e-03

[debug]: error_norm             - function end
[debug]: ~Laplacian_FD          - function begin
```

Figure 9: Debug for verification

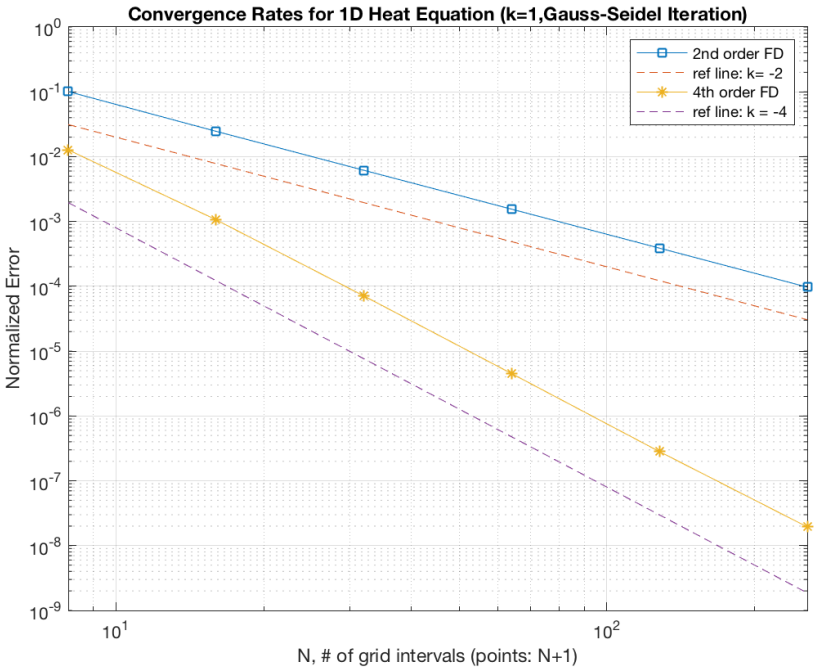


Figure 10: Convergence Rates for 1D Heat Equation with Gauss-Seidel

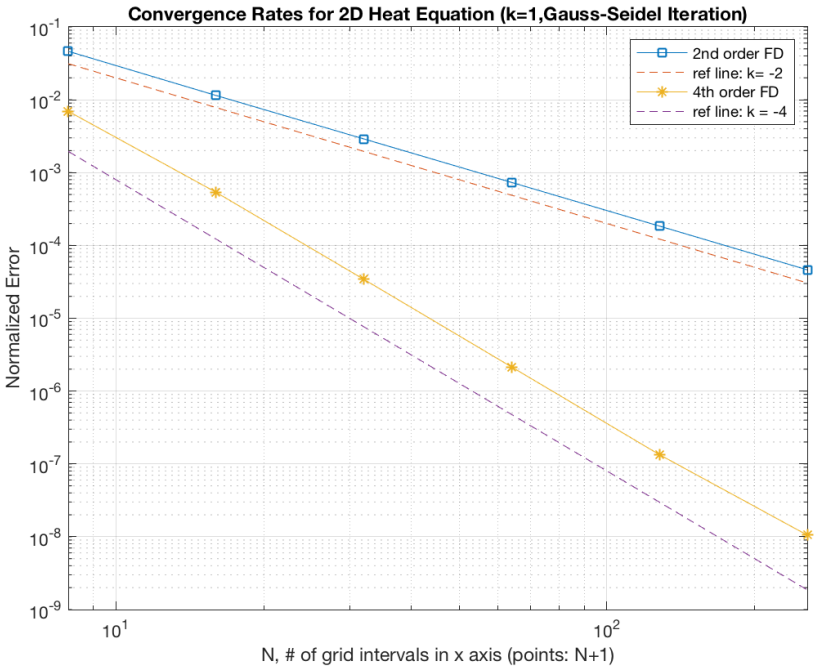


Figure 11: Convergence Rates for 2D Heat Equation with Gauss-Seidel

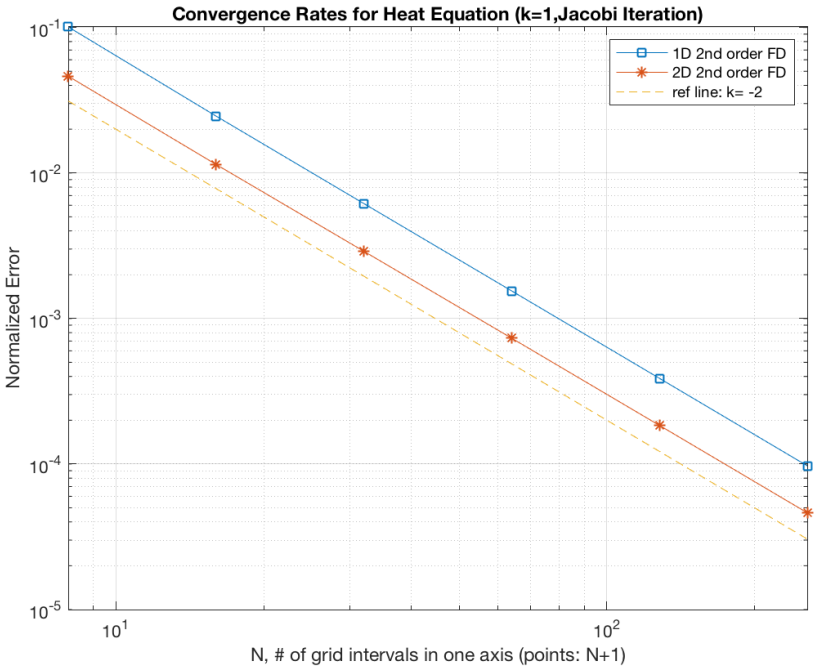


Figure 12: Convergence Rates for Heat Equation with Jacobi

```
-----------------------------------------------------------------------------------------------
Steady Heat Equation Solver - Performance Timings:      |      Mean      Variance       Count
--> parse_input         : 3.30901e-03 secs ( 46.8695 %) | [3.30901e-03  0.00000e+00           1]
--> init                : 1.85800e-03 secs ( 26.3170 %) | [1.85800e-03  0.00000e+00           1]
--> output              : 1.77598e-03 secs ( 25.1553 %) | [1.77598e-03  0.00000e+00           1]
--> solve_system        : 6.48499e-05 secs (  0.9185 %) | [6.48499e-05  0.00000e+00           1]
--> build_linear_system : 3.88622e-05 secs (  0.5505 %) | [3.88622e-05  0.00000e+00           1]
--> GRVY_Unassigned     : 1.33514e-05 secs (  0.1891 %)

    Total Measured Time = 7.06005e-03 secs (100.0000 %)
-----------------------------------------------------------------------------------------------
```

Figure 13: Runtime summary for 8 points

```
-----------------------------------------------------------------------------------------------
Steady Heat Equation Solver - Performance Timings:      |      Mean      Variance       Count
--> solve_system        : 2.02460e-02 secs ( 71.5406 %) | [2.02460e-02  0.00000e+00           1]
--> parse_input         : 3.22199e-03 secs ( 11.3851 %) | [3.22199e-03  0.00000e+00           1]
--> output              : 2.59995e-03 secs (  9.1871 %) | [2.59995e-03  0.00000e+00           1]
--> init                : 2.02799e-03 secs (  7.1660 %) | [2.02799e-03  0.00000e+00           1]
--> build_linear_system : 1.92165e-04 secs (  0.6790 %) | [1.92165e-04  0.00000e+00           1]
--> GRVY_Unassigned     : 1.19209e-05 secs (  0.0421 %)

    Total Measured Time = 2.83000e-02 secs (100.0000 %)
-----------------------------------------------------------------------------------------------
```

Figure 14: Runtime summary for 32 points

```
-----------------------------------------------------------------------------------------------
Steady Heat Equation Solver - Performance Timings:      |      Mean      Variance       Count
--> solve_system        : 3.93279e+01 secs ( 99.8825 %) | [3.93279e+01  0.00000e+00           1]
--> output              : 2.62620e-02 secs (  0.0667 %) | [2.62620e-02  0.00000e+00           1]
--> init                : 9.86218e-03 secs (  0.0250 %) | [9.86218e-03  0.00000e+00           1]
--> build_linear_system : 7.02906e-03 secs (  0.0179 %) | [7.02906e-03  0.00000e+00           1]
--> parse_input         : 3.09014e-03 secs (  0.0078 %) | [3.09014e-03  0.00000e+00           1]
--> GRVY_Unassigned     : 2.05040e-05 secs (  0.0001 %)

    Total Measured Time = 3.93742e+01 secs (100.0000 %)
-----------------------------------------------------------------------------------------------
```

Figure 15: Runtime summary for 256 points

10 PETSC and Runtime Performance Comparison

To use PETSC, you can just change the "iter_method" to 3. I recommend you to use "mpirun" command as mentioned before to make sure it runs well. In Figure 16, we can see that the time for 2D and $N = 128$ is still around $0.5s$, and the main usage is on PETSC_Initialization.

For comparison, in 1D and 2D case, in Figure 17 and Figure 18 respectively, except for some starting points, as N grows, time for Jacobi and Gauss-Seidel grows as well. But for PETSC, the time for GMRES method is always around $0.4s - 0.6s$, and it is not "very stable" depending on the initialization process. In 2D case, it is obivious that Jacobi and Gauss-Seidel arrive around $100s$. But using GMRES to solve cases for big N will save a lot of time with MPI and some preconditioners.

To reproduce the plots, you can run "**batch_time.sh**" to get a directory called "**time_record**" including all time performance results and use "**time_record.m**" in MATLAB to get the plots.

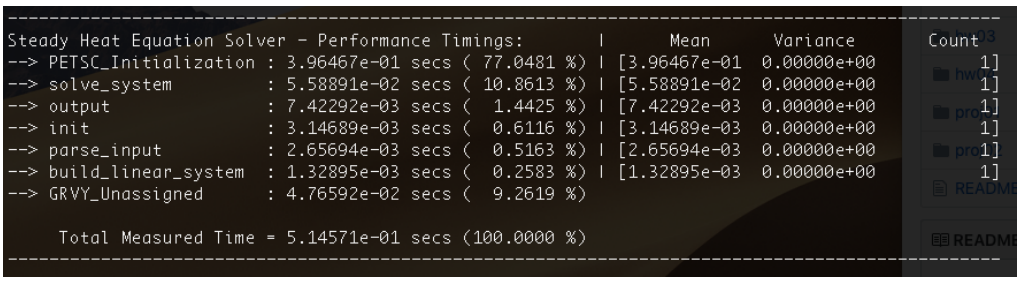


Figure 16: Runtime Performance for PETSC, 2D, 4th order, N = 128

11 Outputs and Tests with HDF5

In the program, the output files are stored in HDF5 files. You can use "**h5dump [Filename]**" to see the datasets. You can use MATLAB, Python, and many tools to read h5 files and to plot with the data. I use MATLAB to plot the results. In test cases, I include all the test cases with reference h4 files and use "**h5diff**" to check the numerical differences.

12 Solution Plots from HDF5

We can visualize the solutions from hdf5 files using MATLAB. We can use "**heat_plot2d.m**" to read data including coordinates: x and y, T_sol and T_exact in hdf5 file from "/test" directory or you can change the path to read the solution you get to visualize it into 3D plot (Figure 20) with "surf" and 2D plot (Figure 21) with "contourf". MATLAB code can also be get from Figure 19:

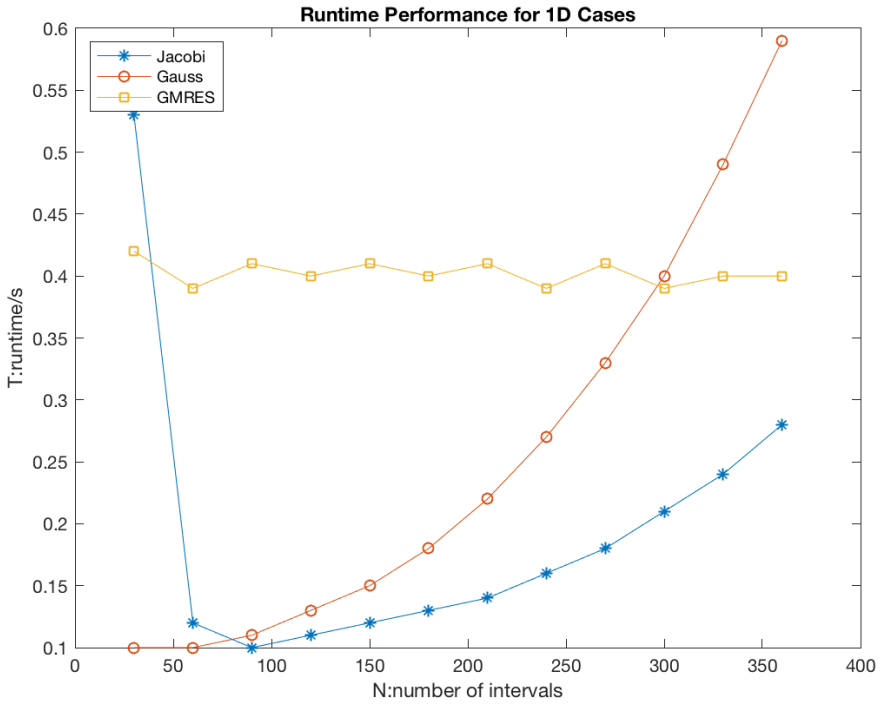


Figure 17: Runtime Performance for 1D Cases

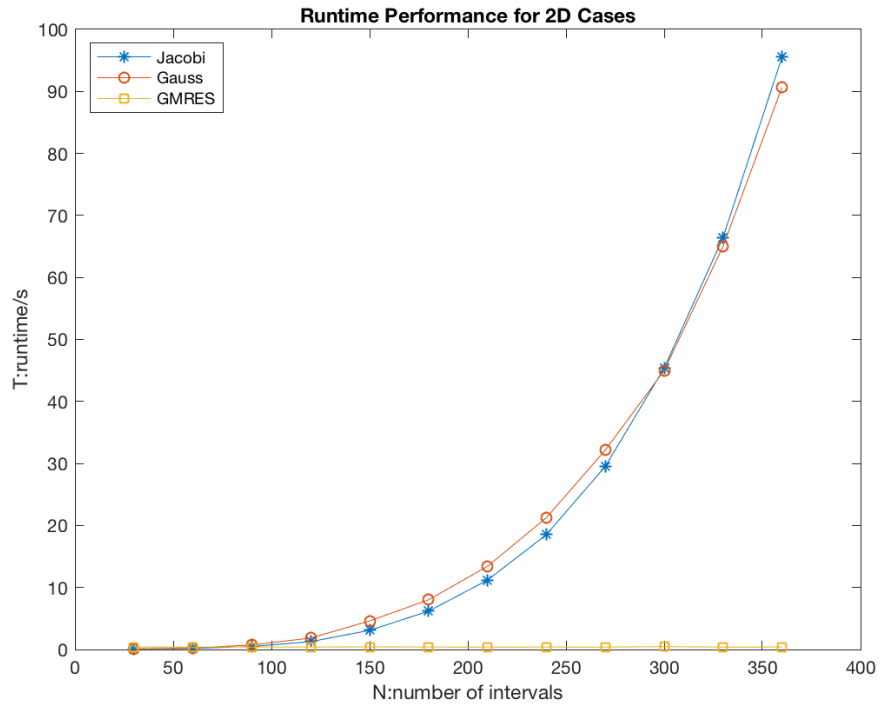


Figure 18: Runtime Performance for 2D Cases

```
clear all;
close all;

% Read dat from h5 file
x = h5read('./test/ref_2d_4th_gauss.h5', '/x');
y = h5read('./test/ref_2d_4th_gauss.h5', '/y');
sol= h5read('./test/ref_2d_4th_gauss.h5', '/T_sol');

% Reshape output data
n = sqrt(length(x));
x = reshape(x,[n,n]);
y = reshape(y,[n,n]);
sol = reshape(sol,[n,n]);

% Plot 3D plot with 2D case using surf
figure
surf(x, y, sol);
colorbar;
title("2D Heat Equation Solution Using Gauss-Seidel, 4th order, N = 128 (surf)");
xlabel('x');
ylabel('y');
zlabel('T\_sol');
saveas(gcf, '2D_plot_surf.png');

% Plot 2D plot with 2D case using contour
figure
contourf(x, y, sol);
colorbar;
title("2D Heat Equation Solution Using Gauss-Seidel, 4th order, N = 128 (contour)");
xlabel('x');
ylabel('y');
saveas(gcf, '2D_plot_contour.png');
```

Figure 19: MATLAB Code for Plotting 2D case

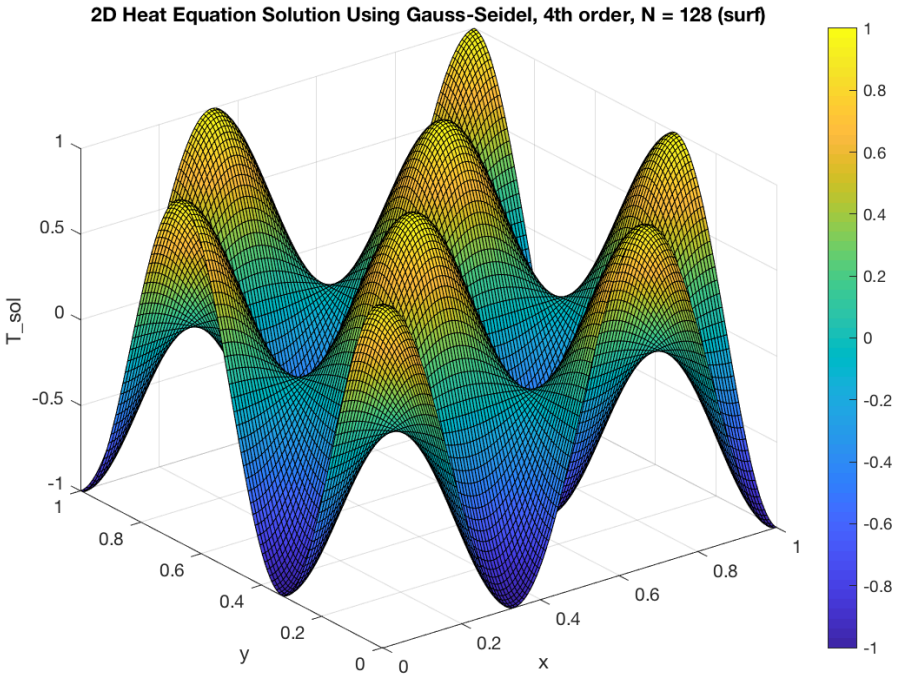


Figure 20: 2D Solution Plot in 3D

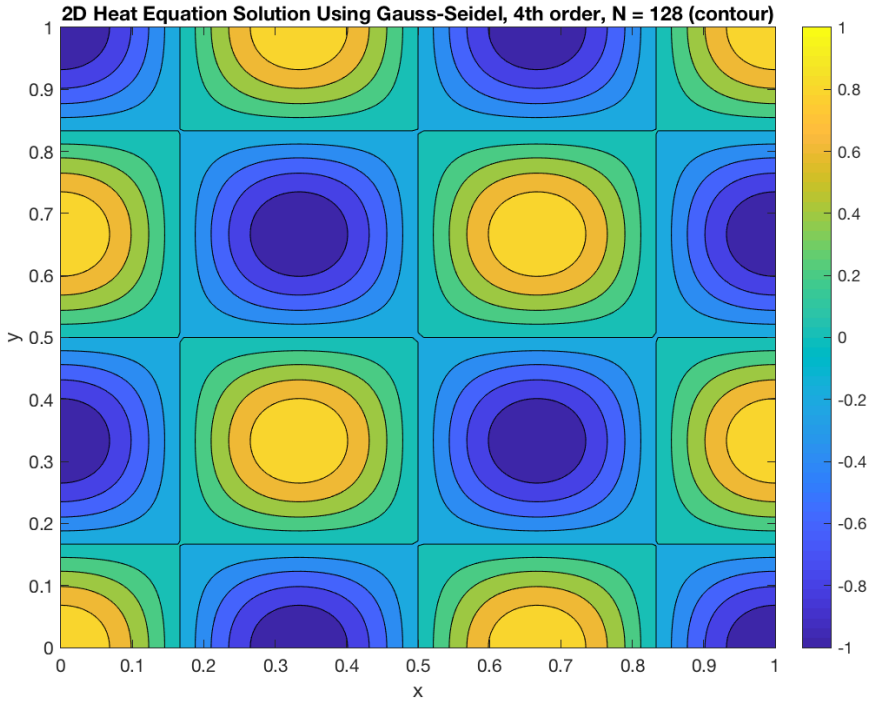


Figure 21: 2D Solution Plot in 2D