ADAPTIVE MESH REFINMENT APPLICATIONS FOR ADVICTION-DIFFUSION PROBLEMS USING AMREX

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Venkata Satya Ramakrishna Raju Kanumuru

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Program:
Computer Science

March 2020

Fargo, North Dakota

North Dakota State University
Graduate School

Title

ADAPTIVE MESH REFINMENT APPLICATIONS FOR ADVICTION-DIFFUSION PROBLEMS USING AMREX

By

Venkata Satya Ramakrishna Raju Kanumuru

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Kendall Nygard
Chair

Dr. Trung Le

Prof. Oksana Myronovych

Approved:

05/15/2020 — Date

Dr. Kendall Nygard — Department Chair

ABSTRACT

In this paper we implemented an adaptive mesh refinement in high performance computing environment to study a wide range of problems in engineering. They are nonlinear Fisher-Kolmogorov equation, heat equation, advection equation and poisons equation using traditional message passing interface (MPI). We used adaptive mesh refinement library called AMReX for computation. AMReX is a numerical library containing the functionality to write massively parallel, block-structured adaptive mesh refinement (AMR) applications. Our study includes examples to solve poisons equation in traditional MPI approach and compared the performance between the two methods.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Trung Le, for his constant guidance throughout the project. His timely feedback and encouraging words helped to complete the paper within time. A special thanks to Center for Computationally Assisted Science and Technology (CCAST), for computational resources. I'd also like to thank my graduate academic advisor, Dr. Kendall Nygard and committee member Prof. Oksana Myronovych, for their support with the study. Lastly, I wish to acknowledge my family and everyone who has been a part of my graduate life in North Dakota State University, for their unfailing support.

DEDICATION

I would like to dedicate this project to my family, friends and teachers who supported me throughout my life.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

In many engineering and science problems, the elliptic equation arises in the form of poisons equation as:

$$\nabla.(\epsilon\nabla\phi) = f \tag{1}$$

Here f and coefficient are given values and φ is derived from boundary conditions. Poisons equation has a broad usage in physics. It is the generalized form of the Laplacian equation. this has many applications like electrostatic potentials or simulation incompressible flows. non-local property of the elliptic equations is important. If is locally dependent, then we can be applied on adaptive mesh refinement using the library presented in this study.

Fast Fourier transforms (FTTs), Cyclic reduction methods, direct sparse solvers, preconditioned iterative methods, multi-grid methods etc. solve the elliptic PDEs. All these methods are differing in supported mesh types or boundary conditions and on coefficient which has to variations like smooth or discontinuous. Time complexity of fastest multi-grid methods is O(N), here n is number of unknowns. whereas time complexity of the FFT methods is around O (N logN). based on parallelization solvers shows significant difference due to the non-local nature of the elliptic equations.

1.1. Heat Equation

More Complicated thing is the extension of the equation (1) can be formulated in a time dependent form. It intensifies the need to find an exact solution for the poisons type equations. Consider two one dimensional diffusion equations, a well-known and famous heat equation.

Heat equation is a partial differential equation that describes how the distribution of some quantity like heat, evolves over time in a solid medium.

$$\frac{\partial\phi}{\partial t} = \nabla^2\phi \tag{2}$$

using forward Euler temporal integration on a periodic domain. We could use a 5-point (in 2D) or 7-point (in 3D) stencil, but for demonstration purposes we spatially discretize the PDE by first constructing (negative) fluxes on cell faces, e.g.,

$$F_{i+1/2,\,j} = \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x} \tag{3}$$

and then taking the divergence to update the cells,

$$\phi_{i,\,j}^{n+1} = \phi_{i,\,j}^{n} + \frac{\Delta t}{\Delta x}\left(F_{i+1/2,\,j} - F_{i-1/2,\,j}\right) + \frac{\Delta t}{\Delta y}\left(F_{i,\,j+1/2} - F_{i,\,j-1/2}\right) \tag{4}$$

1.2. Dirichlet Boundary Condition on Heat Equation

Boundary of a domain takes a value on a solution when applied on partial differential equations or ordinary differential equations is called as Dirichlet boundary condition, Dirichlet boundary condition is also referred as fixed boundary condition.

- ODE

For an ordinary differential equation, for instance,

$$y^{00} + y = 0, \quad y^{00} + y = 0 \tag{5}$$

the Dirichlet boundary conditions on the interval [a, b] take the form

$$y(a) = \alpha, y(b) = \beta, y(a) = \alpha, y(b) = \beta \tag{6}$$

where α and β are given numbers.

- PDE

For a partial differential equation, for example,

$$\nabla^2 y + y = 0, \qquad \nabla^2 y + y = 0, \tag{7}$$

where ∇2 denotes the Laplace operator, the Dirichlet boundary conditions take the form

$$y(x) = f(x) \qquad \forall x \in \partial\Omega, y(x) = f(x) \qquad \forall x \in \partial\Omega, \tag{8}$$

where f is a known function defined on the boundary ∂Ω.

1.3. Fisher–Kolmogorov Equation Solver

Fisher proposed this equation in his 1937 paper the wave of advance of advantageous genes [5] in the context of population dynamics to describe the spatial spread of an advantageous allele and explored its traveling wave solutions.

Muhammad Shaklee has worked on Traveling Wave Solution of the Fisher-Kolmogorov Equation with Non-Linear Diffusion [6]. he chooses the diffusion as a function of cell density such that it is high in highly cell populated areas and low in lower cell populated areas. The Fisher equation with non-linear diffusion is known as modified Fisher equation. Analytical solution for approximation of minimum wave speed by modified Fisher equation using eigenvalues of the stationary states, and numerically by using COMSOL (a commercial finite element solver) is produced in this study. Results shows that minimum wave speed depends on the parameter values. We observe that when diffusion is moderately non-linear, the eigenvalue method correctly predicts the minimum wave speed in our numerical calculations, but when diffusion is strongly non-linear the eigenvalues method gives the wrong answer

And heat equation with nonlinear term called as Fisher-Kolmogorov equation. Fisher and Kolmogorov introduced a classical model to describe the propagation of an advantageous gene in a one-dimensional habitat. The equation describing the phenomenon is a one-dimensional non-linear reaction diffusion equation

Fisher–Kolmogorov equation belongs to the reaction diffusion equation. This equation has extra nonlinear term attached to the heat equation.

$$\frac{\partial \phi}{\partial t} - \frac{\partial^2 \phi}{\partial x^2} = \frac{\alpha}{k}\phi(1-\phi^q) \tag{9}$$

We get, Fisher–Kolmogorov equation Solver as by making constants to unity.

$$\frac{\partial \phi}{\partial t} = \nabla^2 \phi + \phi(1-\phi) \tag{10}$$

Using forward Euler temporal integration on a periodic domain. We could use a 5-point (in 2D) or 7-point (in 3D) stencil, but for demonstration purposes we spatially discretize the PDE by first constructing (negative) fluxes on cell faces, e.g.,

$$F_{i+1/2,\,j} = \frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta x} \tag{11}$$

and then taking the divergence to update the cells,

$$\phi_{i,\,j}^{n+1} = \phi_{i,\,j}^{n} + \frac{\Delta t}{\Delta x}\left(F_{i+1/2,\,j} - F_{i-1/2,\,j}\right) + \frac{\Delta t}{\Delta y}\left(F_{i,\,j+1/2} - F_{i,\,j-1/2}\right) + \Delta t \phi_{i,\,j}^{n}\left(1 + \phi_{i,\,j}^{n}\right) \tag{12}$$

1.4. The Advection Equation

Advection equation is solved on multi-level, adaptive grid structure

$$\frac{\partial \phi}{\partial t} = -\nabla \cdot (\phi \mathbf{U}) \tag{13}$$

The velocity field is a specified divergence-free (so the flow field is incompressible) function of space and time. The initial scalar field is a Gaussian profile. To integrate these equations on a given level, we use a simple conservative update,

$$\frac{\phi_{i,\,j}^{n+1} - \phi_{i,\,j}^{n}}{\Delta t} = \frac{(\phi U)_{i+1/2,\,j}^{n+1/2} - (\phi U)_{i-1/2,\,j}^{n+1/2}}{\Delta x} + \frac{(\phi V)_{i,\,j+1/2}^{n+1/2} - (\phi V)_{i,\,j-1/2}^{n+1/2}}{\Delta y}, \tag{14}$$

Here the velocity U is the parameter that controls the motion of the vortex. Depending on how U varies with space and time, we can have different motion of the vortices. Here we choose the form of U is in equation 12.

1.5. Jacobi Methods

1.5.1. Jacobi nth Order Basic Example

Here the same Jacobi method is applied but the equations are taken through a file input which contains matrices. Here unknowns are given at runtime of the program. This program solves up to ten unknowns. Also, we need to specify the number iterations in run time, or we can

hard code them in the program. I have hard coded it as 200 iterations. As the number of iterations increases the solution converges to exact solution.

$$Ax = b \tag{15}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}. \tag{16}$$

A can be decomposed into a diagonal component D, and the remainder R:

A = D+R where

$$D = \begin{bmatrix} a_{11} & 0 & 0 & \dots & 0 \\ 0 & a_{22} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{nn} \end{bmatrix} \tag{17}$$

$$R = \begin{bmatrix} 0 & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & 0 & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & 0 \end{bmatrix} \tag{18}$$

The solution is then obtained iteratively.

1.5.2. Message Passing Interface Jacobi Iterative Method on Laplace Equation

In many engineering applications, the Laplace equation arises naturally from the governing equations. When the size of the matrix A becomes very large (N = 1 million or billion) then the convergence [8] of the Jacobi method becomes very slow because memory in single CPU is very less to handle large size of matrix. So, we need to accelerate the computation and MPI is the best way to do it.

Linear system of equations is solved using Jacobi iteration [7] of approximations. Laplace equation in two dimensions with finite differences is solved by using this numerical method. relaxation technique is used to find the value of each element. Laplace equation and its

generalized form Poisons equation is the model for relaxation technique. In these equations' boundary values are specified on the boundary of a domain. This technique is used to solve the linear equations resulting from a discretization of the differential equation.

If iterations are converged, then this means diffnorm [8] is less than 1.0e-2. And if the iterations are also specified in the program as 100. MPI is used for communication between the nodes or processors. Each processor is represented based on the rank and communication is done between nodes by the reference of the rank. Here, 12x12 mesh is solved using 4 processors. The processors arrangement looks like the row wise arrangement. The use of ghost points to determine the values in parallel data structure is the simplest. Initial values are given same as the rank in the interior of the mesh and boundary are filled with -1 on top and bottom.

Here I started different variations of the MPI implementations. MPI implementation is done by using the functions called MPI-Send and MPI-Rec. these functions send and receive the data from different neighboring nodes. First thing is applying MPI in column wise by dividing the 12x12 mesh vertically into 4 processors. Later the same is applied on both ways i.e. the division of the rank represent the 2x2 grid with 4 processors here grid means first row has 2 processors and second row has 2 processors. Next application is on the 4x4 grid with 16 processors. Here the number of iterations to find the exact solution is fixed because of the mesh size i.e. 12x12. Later experiment is done on the 2x16 grid with 16 processors on 32x32 mesh. Here I have observed that the iterations are increased to get the exact solution.

1.5.3. Matrix Laboratory Programs Jacobi Iterative Method

Iterative method [7] is also done using the Matrix Laboratory (MATLAB) code in single processor. First done with the simple example and extended to the nth order Jacobi. Also, one-dimensional and two-dimensional poisons equation is also solved using the MATLAB. Showed

the graphs between the exact solution and the solution obtained by the iterative relations technique. The results are showed no difference in the exact and numerical method. as the iterations are increased the convergence of the solution also increased. Please see the appendix for the code.

2. AMREX

Exascale computing refers to computing systems capable of at least one exaflops, or a billionth of billion (i.e. a quintillion) calculations per second. Such capacity represents a thousand-fold increase over the first peta-scale computer that came into operation in 2008. Although the exascale wall for FLOPS was not broken in 2019, the Oak Ridge National Laboratory performed a 1.8x1018 operation calculation per second (which is not the same as 1.8x1018 flops). Exascale computing would be a significant achievement in computer engineering, as an exascale computer would have processing power on the order of the estimated processing power of the human brain at the neural level. AMReX [9] is designed to be used on Exascale computing system. This paper concentrates mainly on AMReX Framework and its applications. Basically, there are two code suites which are based on AMReX Framework [9]. They are AMReXAstro and AMReX-Combustion also called as Pele Suite. AMReX-Astro is a suite of Open astrophysical hydrodynamics codes for exascale architectures and AMReX-Combustion is a suite of adaptive mesh hydrodynamics simulation codes for reacting flows. Also, there are several individual codes based on these AMReX code suites. they are Castro [10], IAMR, MAESTROeX [10], MFIXExa, Nyx, WrapX, PeleLM and PeleC. Castro [10], MAESTROex [10] and Nyx comes under the suit AMReX-Astro [10].

Key features of AMReX Framework are as follows:

- Whole codebase is written in C++ and Fortran interfaces.
- Supports 1-D, 2-D, 3-D.
- Cell-centered, face-centered, edge-centered, and nodal data are supported.
- Elliptic, hyperbolic or parabolic solution on hierarchical adaptive grid structure is supported.

- Optional subcycling in time for time dependent PDEs
- Support for particles
- Support for embedded boundary (cut cell) representations of complex geometries

Amrex follow a simple process to divide large domain into small grids and then distribute small grids to MPI ranks. Gridding and distributing the grids to MPI ranks is combinedly called as the load balancing. Gridding is based on the Berger-Rigoutsis clustering algorithm. Any algorithm from the options like Knapsack, SFC or Round-robin can be used while load balancing.

2.1. Load Balancing

Load balancing process is independent of the grid creation, but weights assigned to the grids and each grid are the inputs for the load balancing. Different algorithms supported by the Amrex are as follows.

Knapsack algorithm: Array of weights, one per grid or MultiFab of weights per cell used to compute the weight per grid can be passed using Amrex.

SFC: this is the default algorithm used in load balancing in Amrex. Enumerate grids with a space -filling Z-Morton curve, then partition the resulting ordering across ranks in a way that balances the load.

Round-robin: Fab I is owned by CPU i%N. where N is total number of MPI ranks.

2.2. Amr-Mesh and Amr-Core

User need to build the Geometry, DistributionMaping and BoxArray objects in single-level simulation. For multiple level simulations AmrMesh class is the container which stores arrays of these objects and information about grid structure. The protected data members are:

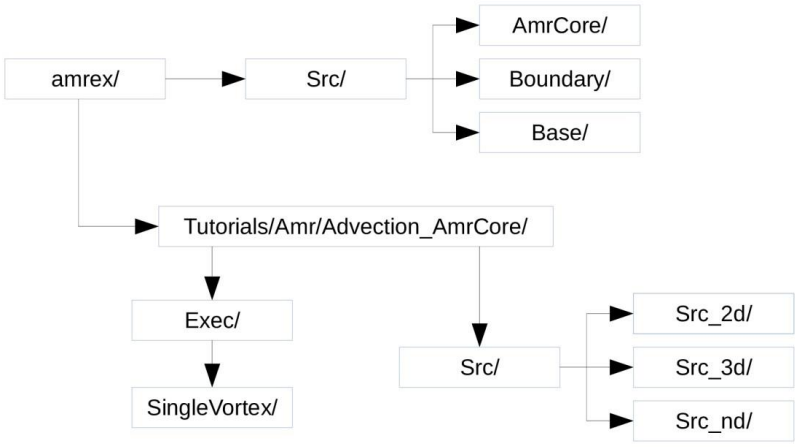

Figure 1. Source Code Tree for The Amradvection Amrcore Example

```
protected:
    int             verbose;
    int             max_level;       // Maximum allowed level.
    Vector<IntVect> ref_ratio;        // Refinement ratios [0:finest_level-1]

    int             finest_level;    // Current finest level.

    Vector<IntVect> n_error_buf;      // Buffer cells around each tagged cell.
    Vector<IntVect> blocking_factor; // Blocking factor in grid generation
                                     // (by level).
    Vector<IntVect> max_grid_size;    // Maximum allowable grid size (by level).
    Real            grid_eff;        // Grid efficiency.
    int             n_proper;        // # cells required for proper nesting.

    bool use_fixed_coarse_grids;
    int  use_fixed_upto_level;
    bool refine_grid_layout;         // chop up grids to have the number of
                                     // grids no less the number of procs

    Vector<Geometry>            geom;
    Vector<DistributionMapping> dmap;
    Vector<BoxArray>            grids;
```

Figure 2. Protected Code

AmrCore is a pure virtual class, which is derived from the AmrMesh class. There are no pure virtual functions in AmrMesh, there are 5 pure virtual functions in AmrCore class. They are as follows. AmrCore have a member functions, most of them override the base class AmrMesh. Applications must implement these 5 functions.

```
//! Tag cells for refinement.  TagBoxArray tags is built on level lev grids.
virtual void ErrorEst (int lev, TagBoxArray& tags, Real time,
                       int ngrow) override = 0;

//! Make a new level from scratch using provided BoxArray and DistributionMapping.
//! Only used during initialization.
virtual void MakeNewLevelFromScratch (int lev, Real time, const BoxArray& ba,
                                      const DistributionMapping& dm) override = 0;

//! Make a new level using provided BoxArray and DistributionMapping and fill
//  with interpolated coarse level data.
virtual void MakeNewLevelFromCoarse (int lev, Real time, const BoxArray& ba,
                                     const DistributionMapping& dm) = 0;

//! Remake an existing level using provided BoxArray and DistributionMapping
//  and fill with existing fine and coarse data.
virtual void RemakeLevel (int lev, Real time, const BoxArray& ba,
                          const DistributionMapping& dm) = 0;

//! Delete level data
virtual void ClearLevel (int lev) = 0;
```

Figure 3. Functions

2.3. Tag-Box and Cluster

These class are used in the grid creation, cells tagged for refinement is marked by a data structure called TagBox. Cluster class helps sort tagged cells and generate a grid structure with tagged cells. Regrid and ErrorEst are the interfaces which hide the member functions.

2.4. Fill-Patch-Util and Interpolater

Array of MultiFabs one for each level of refinement uses the fillpatch to fill MultiFabs temporarily that include different ghost cells based on the coarsest level and non-coarsest level. AMReX FillPatchUtil.cpp contains two functions. FillPatchUtil uses an interpolator. Interpolator is a virtual base class and have some derived classes.

- FillPatchSingleLevels (): this fills the ghost region at single level of refinement. It interpolates in time between tow MultiFabs associated with different times.
- FillPatchTwoLevels (): this fills the ghost regions at single level of refinement, assuming the is a coarse level. This uses the FillPatchSingleLevel for interpolation.

3. LITERATURE REVIEW

Over the last decade block-structured adaptive mesh refinement usage has increased a lot along with the codebases and frameworks. block-structured adaptive mesh refinement frameworks have evolved to different paths. Amrex is one of the frameworks which have been in existence over a decade or more, with reasonably sized and active user base and is publicly available. Earlier Amrex is called with the name called Boxlib. There are various other frameworks available but this amrex is compatible and suitable for the latest hardware systems. There are different other frameworks that are like the amrex. Most of the explicit methods for compressible hydrodynamics are solved using these frameworks in the early times. Some of these frameworks are AstroBEAR, CRASH, Cactus, Enzo, FLASH, Overture, PLUTO, and Uintah [1]. These frameworks focused on specific domain individually. While other frameworks concentrated on more general functionalities, all frameworks can solve hyperbolic conservation laws explicitly but not all frameworks have the functionality to solve elliptic equations accurately. AMROC and AMRClaw frameworks are used to solve g hyperbolic conservation laws specifically, also these latest frameworks include tsunami simulation tool. Amrex, Chombo, Jasmine and SAMRAI are the frameworks, who's main functionality is to solve hyperbolic, parabolic and elliptic equations. PARAMESH is for mesh management which achieves equation independent ability.

Over the last decade block-structured adaptive mesh refinement usage has increased a lot along with the codebases and frameworks. block-structured adaptive mesh refinement frameworks have evolved to different paths. Amrex is one of the frameworks which have been in existence over a decade or more, with reasonably sized and active user base and is publicly

available. Earlier Amrex is called with the name called Boxlib. There are various other frameworks available but this amrex is compatible and suitable for the latest hardware systems.

There are different other frameworks that are like the amrex. Most of the explicit methods for compressible hydrodynamics are solved using these frameworks in the early times. Some of these frameworks are AstroBEAR, CRASH, Cactus, Enzo, FLASH, Overture, PLUTO, and Uintah [1]. These frameworks focused on specific domain individually. While other frameworks concentrated on more general functionalities, all frameworks can solve hyperbolic conservation laws explicitly but not all frameworks have the functionality to solve elliptic equations accurately. AMROC and AMRClaw frameworks are used to solve g hyperbolic conservation laws specifically, also these latest frameworks include tsunami simulation tool. Amrex, Chombo, Jasmine and SAMRAI are the frameworks, who's main functionality is to solve hyperbolic, parabolic and elliptic equations. PARAMESH is for mesh management which achieves equation independent ability.

Amrex is a framework for building massively parallel adaptive mesh refinement. This framework helps developers to implement new algorithms fast. Also, this framework is easy to implement large scale domain specific numerical analysis in the fields like astrophysics, cosmology, surface flow, turbulent combustion and mainly on time dependent PDE's. this framework is not fixed to time stepping or discretization strategy, this is the reason for its flexibility.

On adaptive mesh, AMReX provides very general functionality for solving the time dependent PDE's. Also, there are specific applications like CASTRO and MAESTRO which are based on AMReX framework are used to solve fully compressible radiation-hydrodynamics and low Mach number astrophysical flows, respectively. My work is like the kind of study is done by

P. M. Ricker on a direct multigrid poison solver for oct-tree adaptive meshes using PARAMESH library [2]. This study is done by a finite volume method for solving poison equation on oct-tree adaptive meshes. This is the modified method of the Huang and Greengard's method which is based on the finite differences meshes and refined patches doesn't share boundaries. Ricker work uses the FLASH code framework and also the PARAMESH library.

J. Teunissen and R. Keppens has done a similar work on a geometric multigrid library for quadtree/octree AMR grids coupled to MPI-AMRVAC [3]. Their geometric multigrid library is efficient MPI-parallel library for quadtree(2D) or octree(3D) grids with adaptive refinement. Second order discretization for elliptic operators with cartesian 2D/3D and cylindrical 2D geometries are supported. Boundary conditions like Periodic, Dirichlet, and Neumann can be handled. FFT-based solver on the coarse grid is used for free-space boundary conditions for 3D poisons problems. Scaled results up to 1792 cores using this library are showed. This library extended the MPI-AMRVAC an adaptive mesh refinement framework with an elliptic solver. divergence of the magnetic field in magnetohydrodynamic simulations is controlled by the multigrid routines for several test cases.

Poisson solvers in the unit cube are used in many applications in computational science and engineering. The fast Fourier transform (FFT), the fast multipole method (FMM), the geometric multigrid (GMG), and algebraic multigrid (AMG) are widely used methods for solving poisons solver in the unit cube. In this paper high order, highly nonuniform discretization's is the main focus among solvers. FFT and regular-stencil multigrid are the solvers which are specialized for problems on regular grids. high-performance geometric multigrid (HPGMG) is a finite element multigrid benchmark is used to here. Five codes are shown in this study, in which three methods are developed in their group. FFT, GMG, and FMM

are parallel solvers, these use high-order approximation schemas for poison problems with continuous forcing function (right hand side). These are based on FFTW for single node parallelism, AMG code is from the Trillions library from the Sandia National Laboratory [4], GMG and FMM support octree-based mesh refinement and variable coefficients and enable highly nonuniform discretization's. results also considered weak scaling, strong scaling, and time to solution for uniform and highly refined grids. Stampede and Titan are two supercomputer systems used to test these solvers. 600 billion unknowns on 229,379 cores of Titan is the largest test case [4]. Their results show smooth source functions that require uniform resolution are good with FFT. When source function is considered with internal sharp layers then FFT is less efficient when compared to the FMM and GMG, which showed high sensitivity to quality. The low-order accurate counterparts are less efficient when compared with high-order accurate versions of GMG and FMM [4].

Alzheimer's disease is an irreversible neuro degenerative disorder that manifests itself in the progressive aggregation of misfolded tau protein, neuronal death, and cerebral atrophy.

4. RESULTS

4.1. Heat Equation

Heat equation is a partial differential equation that describes how the distribution of some quantity like heat, evolves over time in a solid medium.

$$\phi_{i,j}^{n+1} = \phi_{i,j}^{n} + \frac{\Delta t}{\Delta x}\left(F_{i+1/2,j} - F_{i-1/2,j}\right) + \frac{\Delta t}{\Delta y}\left(F_{i,j+1/2} - F_{i,j-1/2}\right) \tag{19}$$

Initial conditions of heat source is located at x initial = 0.25 and y initial = 0.25, and the boundary condition used is periodic, Now we use the heat equation 6 to simulate the propagation of heat inside this domain, in fig.1 heat is propagating in all directions along with the x-y coordinates, at time T = 0.0274658 and T = 0.0961304 and finally at T = 0.1373290.

The implementation details of the code are discussed in section Example: Heat Equation EX1 C For now let's visualize the results.

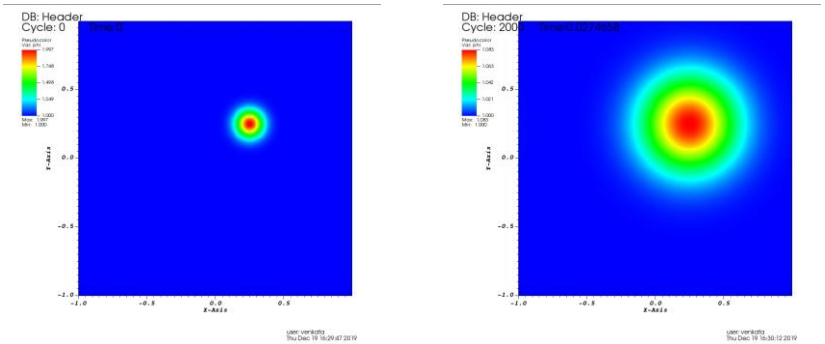

Figure 4. Showing Heat Expansion with Time and Space T=0.0000000 & T=0.0274658

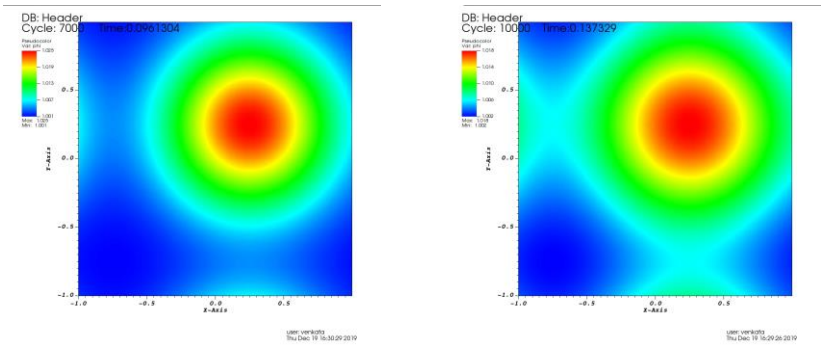

Figure 5. Showing Heat Expansion with Time and Space T=0.0961304 & T=0.1373290

4.2. Dirichlet Boundary Condition on Heat Equation

Input is given in the form of file and then this input is parsed with param function to take values for boundary type. Here we are taking Dirichlet boundary condition as INLET (phi = val at boundary) allowable options for this example are

-1 = PERIODIC

11 = INLET (phi = val at boundary)

12 = OUTLET (phi = extrap at boundary)

14 = SLIP WALL (dphi/dn=0 at boundary)

15 = NO SLIP WALL (dphi/dn=0 at boundary)

bcx lo = 11

bcx hi = 11

bcy lo = 11

bcy hi = 11

Main function which solves heat equation calls a subroutine named multifa fill ghost cells which in-turn calls another subroutine named multifab physbc. this multifab physbc is where we give our boundary conditions a value. where we specifically mention the type of the boundary condition in the input file.

```
!-------------------------------------------------------------------------------
! lower X
!-------------------------------------------------------------------------------
if (bc(1,1) == EXT_DIR) then
   ! set all ghost cell values to a prescribed dirichlet
   ! value; in this example, we have chosen 1
   do j = lo(2)-ng, hi(2)+ng
      s(lo(1)-ng:lo(1)-1,j) = 2 ! or any value to put on boundary
   end do
!-------------------------------------------------------------------------------
! upper X
!-------------------------------------------------------------------------------
if (bc(1,2) == EXT_DIR) then
   do j = lo(2)-ng, hi(2)+ng
      s(hi(1)+1:hi(1)+ng,j) = 2 ! or any value to put on boundary
   end do
!-------------------------------------------------------------------------------
! lower Y
!-------------------------------------------------------------------------------
if (bc(2,1) == EXT_DIR) then
   do i = lo(1)-ng, hi(1)+ng
      s(i,lo(2)-ng:lo(2)-1) = 2 ! or any value to put on boundary
   end do
!-------------------------------------------------------------------------------
! upper Y
!-------------------------------------------------------------------------------
if (bc(2,2) == EXT_DIR) then
   do i = lo(1)-ng, hi(1)+ng
      s(i,hi(2)+1:hi(2)+ng) = 2 ! or any value to put on boundary
   end do
```

Figure 6. Boundary Conditions

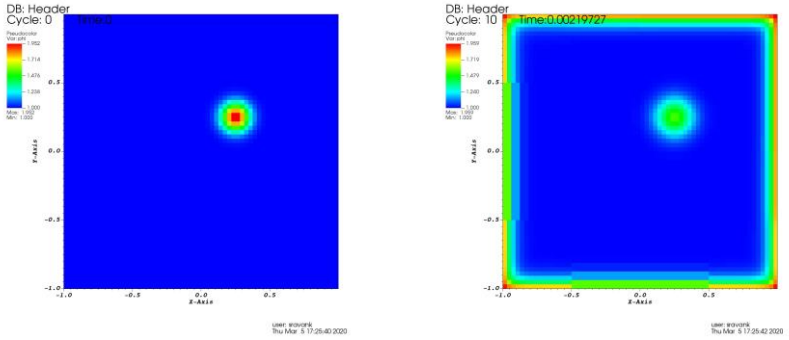

Figure 7. Dirichlet Boundary Condition on Heat Equation with Time and Space T=0.000000 & T=0.00219727

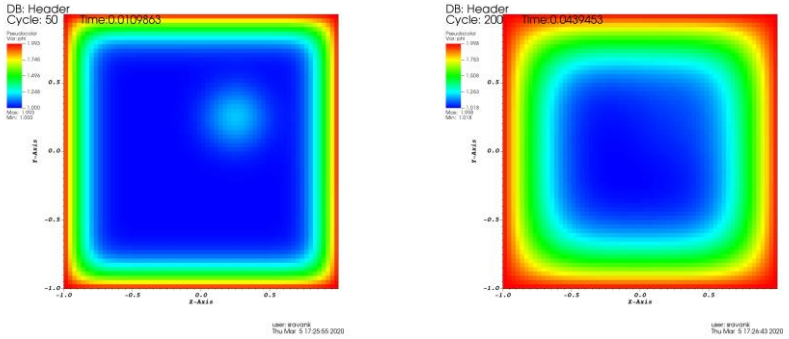

Figure 8. Dirichlet Boundary Condition on Heat Equation with Time and Space T=0.01098630 & T=0.04394530

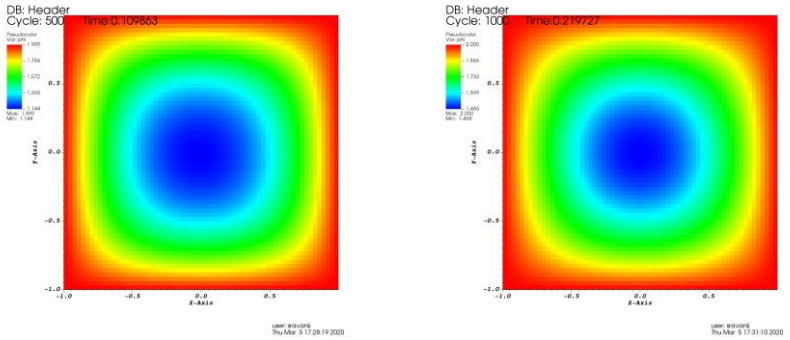

Figure 9. Dirichlet Boundary Condition on Heat Equation with Time and Space T=0.10986300 & T=0.21972700

4.3. Fisher–Kolmogorov Equation Solver

Fisher–Kolmogorov equation belongs to the reaction diffusion equation. This equation has extra nonlinear term attached to the heat equation.

$$\phi_{i,j}^{n+1} = \phi_{i,j}^{n} + \frac{\Delta t}{\Delta x}\left(F_{i+1/2,j} - F_{i-1/2,j}\right) + \frac{\Delta t}{\Delta y}\left(F_{i,j+1/2} - F_{i,j-1/2}\right) + \Delta t \phi_{i,j}^{n}\left(1 + \phi_{i,j}^{n}\right) \tag{20}$$

Initial conditions of wave source is located at x initial = 0.0 and y initial = 0.0, and the boundary condition used is periodic, Now we use the wave equation 10 and 11 to simulate the propagation of heat inside this domain, in fig.2 wave is propagating in all directions along with the x-y coordinates, at time T = 0.0274658 and T = 0.0961304 and finally at T = 0.1373290.

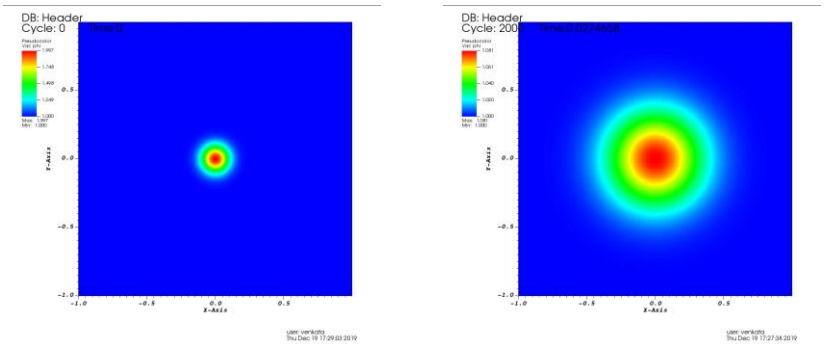

Figure 10. Fisher Kolmogorov Wave Equation with Time and Space T=0.0000000 & T=0.0274658

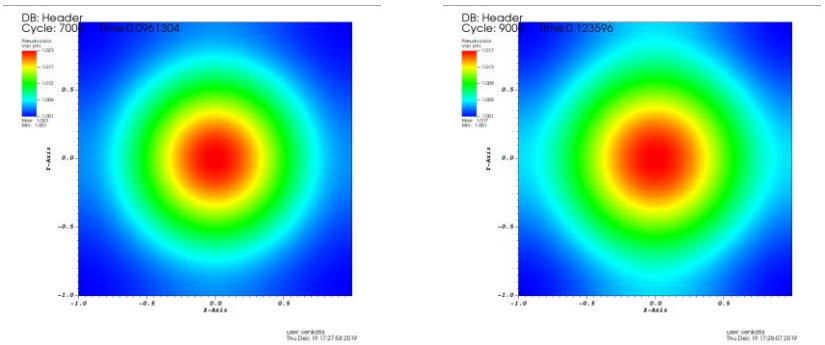

Figure 11. Fisher Kolmogorov Wave Equation with Time and Space T=0.0961304 & T=0.1373290

4.4. The Advection Equation

Advection equation is solved on multi-level, adaptive grid structure. Initialization of velocity field is done on different vortex's and each formula is shown below. Also, the boundary condition used is periodic.

Single Vortex

$$r2 = \frac{(x-0.5)^2 + (y-0.75)^2}{0.01} \tag{21}$$

$$\phi_{i,j,k} = 1.0 + e^{-r2} \tag{22}$$

Double Vortex

$$r2 = \frac{(x-0.5)^2 + (y-0.75)^2}{0.01} \tag{23}$$

$$r21 = \frac{(x-0.25)^2 + (y-0.5)^2}{0.01} \tag{24}$$

$$\phi_{i,j,k} = 1.0 + e^{-r2} + e^{-r21} \tag{25}$$

Triple Vortex

$$r2 = \frac{(x-0.5)^2 + (y-0.75)^2}{0.01} \tag{26}$$

$$r21 = \frac{(x-0.25)^2 + (y-0.5)^2}{0.01} \tag{27}$$

$$r22 = \frac{(x-0.75)^2 + (y-0.5)^2)}{0.01} \tag{28}$$

$$\phi_{i,j,k} = 1.0 + e^{-r2} + e^{-r21} + e^{-r22} \tag{29}$$

Quadra Vortex

$$r2 = \frac{(x-0.5)^2 + (y-0.75)^2}{0.01} \tag{30}$$

$$r21 = \frac{(x-0.25)^2 + (y-0.5)^2}{0.01} \tag{31}$$

$$r22 = \frac{(x-0.75)^2 + (y-0.5)^2)}{0.01} \tag{32}$$

$$r23 = \frac{(x-0.75)^2 + (y-0.5d0)^2)}{0.01} \tag{33}$$

$$\varphi i,j,k = 1.0 + e{-}r2 + e{-}r21 + e{-}r22 \tag{34}$$

Below figures shows the differences between the multiple vortices at Time = 0 sec and at Time = 0.480124

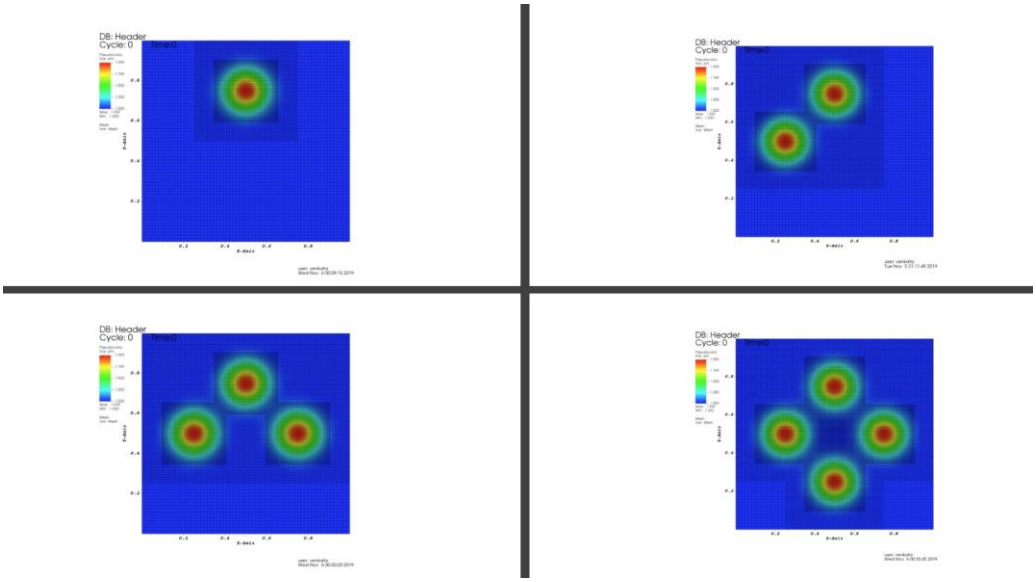

Figure 12. 2D Multiple Vortex Comparison Time = 0 sec

The impact of U on the motion of the vortices is described in this case, the adaptive mesh refinement plays a significant role in solving the equations from 14 to 25 because they follow the motion of the vortices of figures 3 and 4 by comparing the mesh. The benefit of having adaptive mesh is to solve the vortices at location the source exists.

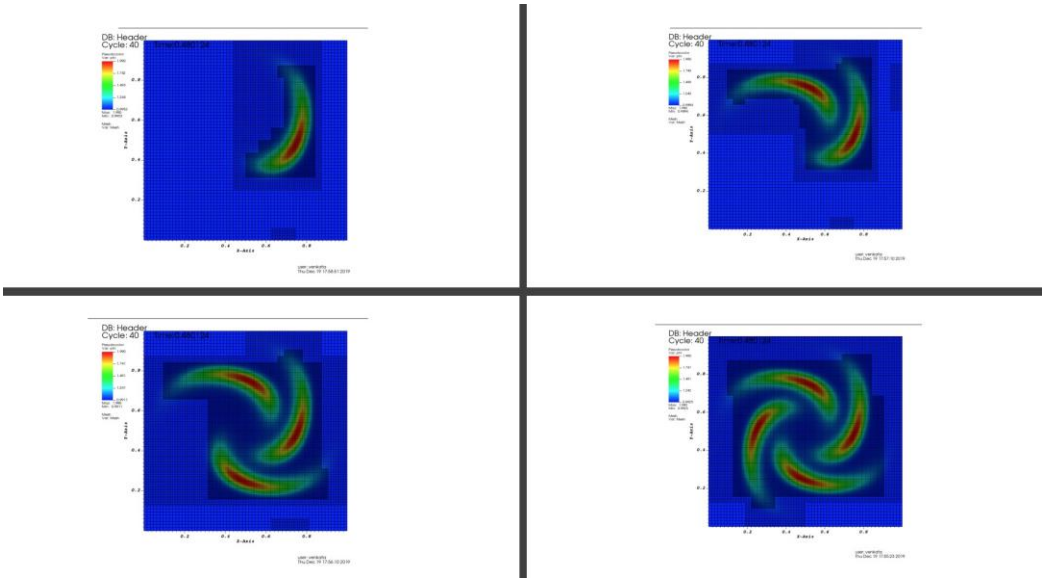

Figure 13. 2D Multiple Vortex Comparison Time = 0.480124 sec

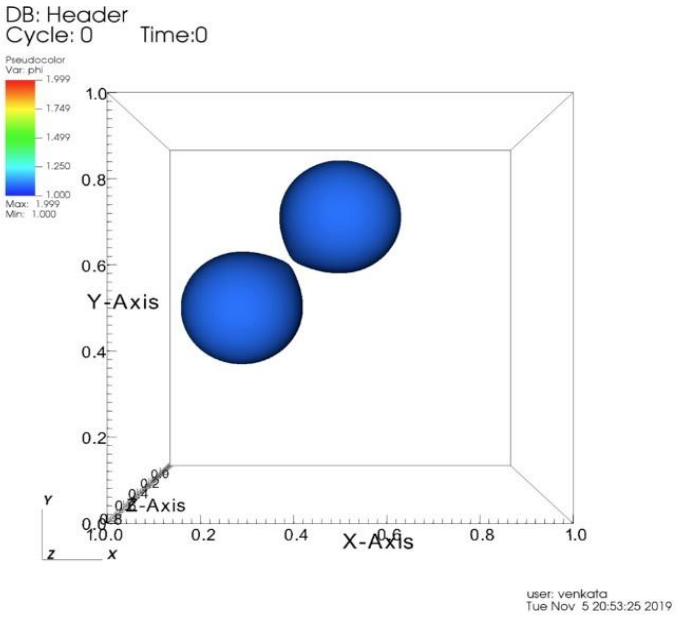

Figure 14. 3D Double Vortex with Time = 0 sec

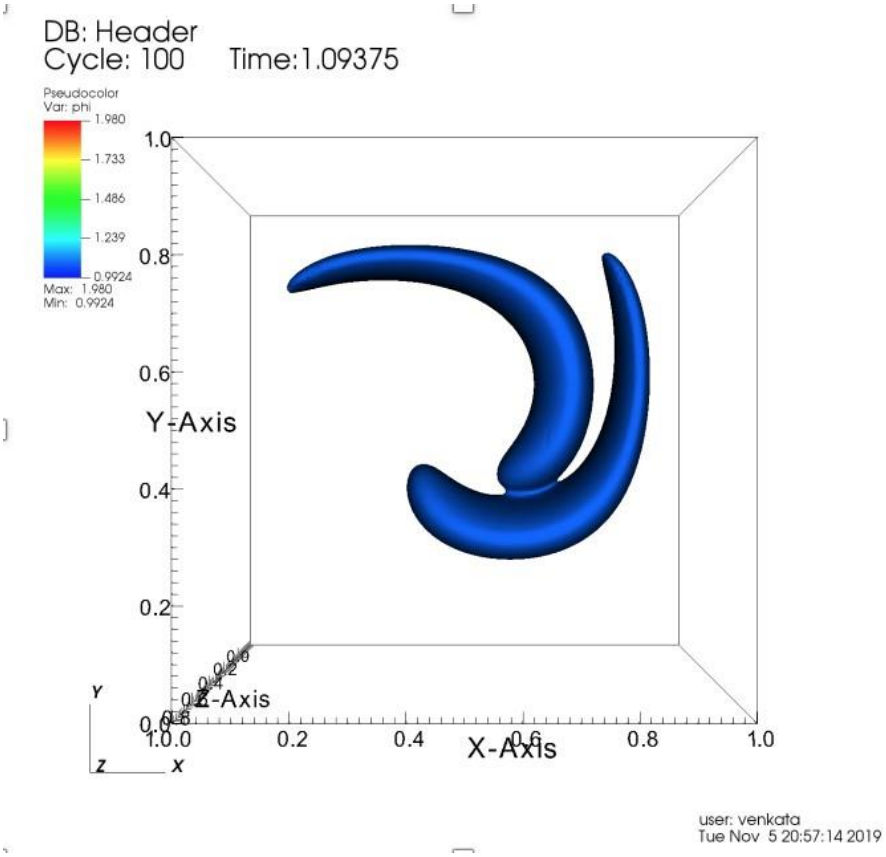

Figure 15. 3D Double Vortex with Time = 1.09375 sec

5. CONCLUSION

This paper presents massively parallel, block-structured adaptive mesh refinement (AMR) framework (AMReX) to solve various applications. First heat equation is solved using this framework and next an extension to the heat equation by adding nonlinear term which gives a reaction diffusion equation called Fisher–Kolmogorov equation. Also showed how heat and wave propagates in 2D and 3D. Later we solved Advection equation using AMReX Framework. Also showed Multiple vortex comparison with respect to time in 2D and 3D environments. And finally solved the MPI based parallel programming example using Jacobi iterative methods.

REFERENCES

[1] Anshu Dubey, Ann Almgren, John Bell, Martin Berzins, Steve Brandt, Greg Bryan, Phillip Colella, Daniel Graves, Michael Lijewski, Frank Löffler, Brian O'Shea, Erik Schnetter, Brian Van Straalen, Klaus Weide, A survey of high level frameworks in block-structured adaptive mesh refinement packages, Journal of Parallel and Distributed Computing, Volume 74, Issue 12, 2014, Pages 3217-3227, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2014.07.001.

[2] Ricker, Paul. (2007). A Direct Multigrid Poisson Solver for Oct-Tree Adaptive Meshes. Astrophysical Journal, Supplement Series. 176. 10.1086/526425.

[3] J. Teunissen, R. Keppens, A geometric multigrid library for quadtree/octree AMR grids coupled to MPI-AMRVAC, Computer Physics Communications, Volume 245, 2019, 106866, ISSN 0010-4655, https://doi.org/10.1016/j.cpc.2019.106866.

[4] Malhotra, Dhairya & Sundar, Hari & Biros, George. (2014). FFT, FMM, or Multigrid? A comparative study of state-of-the-art Poisson solvers. SIAM Journal on Scientific Computing (submitted).

[5] Fisher, R.A. (1937), The wave of advance of advantageous genes. Annals of Eugenics, 7: 355-369. doi:10.1111/j.1469-1809.1937.tb02153.x

[6] Shakeel, Muhammad. (2013). Travelling Wave Solution of the Fisher-Kolmogorov Equation with Non-Linear Diffusion. Applied Mathematics. 04. 148-160. 10.4236/am.2013.48A021.

[7] MPI Forum. Argonne National Laboratory, A simple Jacobi iteration, https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobi/C/main.html

[8] HarpinderKaur, HarpinderKaur. (2012). Convergence of Jacobi and Gauss-Seidel Method and Error Reduction Factor. IOSR Journal of Mathematics. 2. 20-23. 10.9790/5728-0222023.

[9] Zhang, Weiqun & Almgren, Ann & Beckner, Vince & Bell, John & Blaschke, Johannes & Chan, Cy & Day, Marcus & Friesen, Brian & Gott, Kevin & Graves, Daniel & Katz, Max & Myers, Andrew & Nguyen, Tan & Nonaka, Andrew & Rosso, Michele & Williams, Samuel & Zingale, Michael. (2019). AMReX: a framework for block-structured adaptive mesh refinement. Journal of Open Source Software. 4. 1370. 10.21105/joss.01370.

[10] Zingale, M. & Almgren, Ann & Sazo, M. & Beckner, V. & Bell, John & Friesen, B. & Jacobs, A. & Katz, M. & Malone, Chris & Nonaka, Andy & Willcox, D. & Zhang, W.. (2017). Meeting the Challenges of Modeling Astrophysical Thermonuclear Explosions: Castro, Maestro, and the AMReX Astrophysics Suite. Journal of Physics: Conference Series. 1031. 10.1088/1742-6596/1031/1/012024.

APPENDIX A. JACOBI ITERATIVE METHOD WITH BASIC EXAMPLE

Numerical methods most commonly involve iterative methods and Jacobi iterative method is one of them. I am experimenting on the various equations by applying the Jacobi iterative method. Here in this example I took three equations with three unknowns and arranged in matrix form Ax = b. where x is the unknown matrix of size 3x1, A is the matrix of size 3x3 and with each coefficient of the left hand side of the equations and b is the matrix with 3x1 order which represent the right hand side of the equations. Also, by the assumption of the initial unknown values as 0's and applying on the new unknown values derived from the Jacobi method we get the new unknown values. These new unknown values are applied iteratively on the Jacobi formula. After multiple iterations we can see the convergence of the unknown values to the solution.

Ax = b

Basic Example

$$\begin{bmatrix} 4 & 2 & 3 \\ 3 & -5 & 2 \\ -2 & 3 & 8 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8 \\ -14 \\ 27 \end{bmatrix} \tag{A1}$$

we can solve the first row for x, the second for y and the third for z

$$x = (8 - 2y - 3z)/4 \tag{A2}$$

$$y = (-14 - 3x - 2z)/(-5) \tag{A3}$$

$$z = (27 + 2x - 3y)/8 \tag{A4}$$

we will begin with x = y = z = 0 as our initial approximation

$$x = (8 - 2 * 0 - 3 * 0)/4 = 2 \tag{A5}$$

$$y = (-14 - 3 * 0 - 2 * 0)/(-5) = 2.8 \tag{A6}$$

$$z = (27 + 2 * 0 - 3 * 0)/8 = 3.375 \tag{A7}$$

Although the early iterations do not look promising in terms of convergence, things do eventually settle down.

APPENDIX B. JACOBI ITERATIVE METHOD WITH BASIC EXAMPLE

```
#include <stdio.h>

int main ()
{
	int a[3][3] = {{4, 2, 3}, {3, -5, 2}, {-2, 3, 8}}; int b[3][1] = {{8}, {-14},
	{27}}; int i, j; float x1 = 0, x2 = 0, x3 = 0, x1_new, x2_new,
	x3_new;

	for (i = 1; i < 100; i++)
	{ x1_new = (b[0][0] - a[0][1] * x2 - a[0][2] * x3) / a[0][0]; x2_new = (b[1][0] -

		a[1][0] * x1 - a[1][2] * x3) / a[1][1]; x3_new = (b[2][0] - a[2][0] * x1 -

		a[2][1] * x2) / a[2][2];

		x1 = x1_new;
		x2 = x2_new;
		x3 = x3_new; }
		int c[3][1] =
		{{x1}, {x2},
		{x3}}; for (i = 0;
		i < 3; i++)
	{
		j = 0;
		printf ("%d \n\n", c[i][j]);
	}
}
```

Commands to compile and run:

```
$ mpicc -o jacobi jacobi.c
$ mpirun ./jacobi
```

APPENDIX C. JACOBI NTH ORDER BASIC EXAMPLE

Ax=B is the equation and we need to give the order of the matrix. A matrix will be given in a file and its order is NxN and X will be Nx1 and B will be Nx1.

Input file names are matrix_a.csv, matrix_x.csv and matrix_b.csv.

```
#include <stdio.h>

float coeff[10][10]; float Dinv[10][10]; float approx[10][1];
float R[10][10]; //declare the relevant matrices float
matrixRes[10][1]; float b[10][1]; float temp[10][1]; int row,
column, size, navigate;
void multiply(float matrixA[][10], float matrixB[][1])
{ int ctr, ictr;

      //function to perform multiplication for (ctr =
      0; ctr < size; ctr++)
      { matrixRes[ctr][0] = 0;
            for (navigate = 0; navigate < size; navigate++) matrixRes[ctr][0] = matrixRes[ctr][0] +
                  matrixA[ctr][navigate] * matrixB[navigat
      } }
int main()
{
printf("En
ter the
number
of
unknown\
n");
scanf("%d
", &size);
//enter
the size
int
p=8*size;

      FILE *myFile, *myFile1, *myFile2; myFile =
      fopen("matrix_a.csv", "r"); myFile1 =
      fopen("matrix_x.csv", "r"); myFile2 =
      fopen("matrix_b.csv", "r"); if (myFile == NULL)
      { printf("\n file opening failed ");
      } printf("coefficent matrix scanned...\n"); for (row = 0; row <
      size; row++) for (column = 0; column < size; column++)
      fscanf(myFile, "%f,", &coeff[row][column]);
      // for(row=0;row<size;row++)
      //             for(column=0;column<size;column++)
      //                         printf(" coeff %f", coeff[row][column]);
```

```
	printf("first approximation set to 0...\n"); for (row = 0;
	row < size; row++)
		fscanf(myFile1, "%f,", &approx[row][0]);
	// for(row=0;row<size;row++)
	//			printf(" app %f", approx[row][0]);

	printf("RHS coefficient scanned...\n"); for (row = 0;
	row < size; row++) fscanf(myFile2, "%f,",
	&b[row][0]);
	// for(row=0;row<size;row++)
	//			printf(" b %f", b[row][0]);

	for (row = 0; row < size; row++) //We calculate the diagonal inverse matrix make all oth for (column = 0;
		column < size; column++)
		{ if (row == column)
				Dinv[row][column] = 1 / coeff[row][column]; else
				Dinv[row][column] = 0;
		}
	for (row = 0; row < size; row++) for (column = 0; column < size; column++) //calculating the R
		matrix L+U
		{ if (row == column)
			R[row][column] = 0;
			else if (row !=
			column)
				R[row][column] = coeff[row][column];
		}
	int iter;
	// printf("Enter the number of iterations:\n"); //
	scanf("%d",&iter);//enter the number of iterations iter = 200; int
	ctr = 1; int octr;

	while (ctr <= iter)
	{ multiply(R, approx); //multiply L+U and the approximation for (row = 0; row < size;
		row++) temp[row][0] = b[row][0] - matrixRes[row][0]; //the matrix(b-Rx)
		multiply(Dinv, temp);	//multiply D inverse and (b-Rx) for (octr = 0; octr < size; octr++) approx[octr][0]
		= matrixRes[octr][0]; //store matrixRes value in the next approx
		printf("The Value after iteration %d is\n", ctr); for (row = 0; row < size; row++)
		printf("%.3f\n", approx[row][0]); //display the value after the pass
		ctr++;
	}
}
```

Commands to compile and run:

```
$ mpicc -o jacobi jacobi.c
$ mpirun ./jacobi
```

APPENDIX D. MPI JACOBI ITERATIVE METHOD ON LAPLACE EQUATION ROW WISE

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
/* This example handles a 12 x 12 mesh, on 4 processors only. */
#define maxn 12 int main(argc,
argv) int argc; char **argv;
{ int rank, value, size, errcnt, toterr, i, j, itcnt;
      int i_first, i_last;
      MPI_Status status; double diffnorm,
      gdiffnorm; double xlocal[(12 / 4) +
      2][12]; double xnew[(12 / 3) + 2][12];
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      if (size != 4)
            MPI_Abort(MPI_COMM_WORLD, 1);
      /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */ /* Note that top
      and bottom processes have one less row of interior points */ i_first = 1; i_last =
      maxn / size; if (rank == 0) i_first++;
      if (rank == size - 1) i_last--;
      /* Fill the data as specified */ for (i = 1; i
      <= maxn / size; i++)
            for (j = 0; j < maxn; j++) xlocal[i][j] = rank;
      for (j = 0; j < maxn; j++)
      { xlocal[i_first - 1][j] = -2; xlocal[i_last +
            1][j] = -2;
      }
      for (i = 0; i <= 4; i++)
      {
            for (j = 0; j <= 4; j++)
            { printf("%f ", xlocal[i][j]);
            } printf("\n");
      } itcnt = 0;
      do {
            /* Send up unless I'm at the top, then receive from below */ /* Note the
            use of xlocal[i] for &xlocal[i][0] */ if (rank < size - 1)
                        MPI_Send(xlocal[maxn / size], maxn, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD);
            if (rank > 0)
                  MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD, &status);
            /* Send down unless I'm at the bottom */ if (rank >
            0)
                  MPI_Send(xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
                        MPI_COMM_WORLD);
            if (rank < size - 1)
```

```
        MPI_Recv(xlocal[maxn / size + 1], maxn, MPI_DOUBLE, rank + 1, 1,
    MPI_COMM_WORLD, &status); /* Compute new values (but not on boundary) */
    itcnt++; diffnorm = 0.0; for (i = i_first; i <= i_last; i++)
        for (j = 1; j < maxn - 1; j++)
        { xnew[i][j] = (xlocal[i][j + 1] + xlocal[i][j - 1] +
                                    xlocal[i + 1][j] + xlocal[i - 1][j]) /
                                   4.0; diffnorm += (xnew[i][j] -
            xlocal[i][j]) *
                                        (xnew[i][j] - xlocal[i][j]);
        }
    /* Only transfer the interior points */ for (i =
    i_first; i <= i_last; i++) for (j = 1; j < maxn - 1;
    j++) xlocal[i][j] = xnew[i][j];
    MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                    MPI_COMM_WORLD);
    gdiffnorm = sqrt(gdiffnorm); if (rank == 0) printf("At iteration
    %d, diff is %e\n", itcnt, gdiffnorm);
} while (gdiffnorm > 1.0e-2 && itcnt < 100); FILE *fp;
char filename[80]; sprintf(filename,
"Aprocessor%d.txt", rank); if ((fp = fopen(filename,
"wb")) == NULL)
{ printf("Cannot open file.\n");
} for (i = i_first; i <= i_last; i++)
{
    for (j = 1; j < maxn - 1; j++)
    { fprintf(fp, "%f ", xlocal[i][j]);
    } fprintf(fp, "\n");
} fclose(fp);
MPI_Finalize
(); return 0;
}
```

Commands to compile and run:

```
$ mpicc -o row_wise row_wise.c
$ mpirun -np 16 ./row_wise
```

APPENDIX E. MPI JACOBI ITERATIVE METHOD ON LAPLACE EQUATION

COLUMN WISE

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
/* This example handles a 12 x 12 mesh, on 4 processors only. */
#define maxn 12 int main(argc,
argv) int argc; char **argv;
{ int rank, value, size, errcnt, toterr, i, j, itcnt; int j_left, j_right;
      MPI_Status status; double diffnorm, gdiffnorm; double
      xlocal[12][(12 / 4) + 2]; double xnew[12][(12 / 3) + 2];
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      if (size != 4)
            MPI_Abort(MPI_COMM_WORLD, 1);
      /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */
      /* Note that top and bottom processes have one less row of interior points */ j_left
= 1; j_right = maxn / size; if (rank == 0) j_left++;
      if (rank == size - 1) j_right--;
      /* Fill the data as specified */ for (i = 0; i <
      maxn ; i++) for (j = 1; j <= maxn/size; j++)
      xlocal[i][j] = rank;
      for (i = 0; i < maxn; i++)
      { xlocal[i][j_left-1] = -1;
            xlocal[i][j_right+1] = -1;
      } itcnt = 0;
      do {
            /* Send up unless I'm at the top, then receive from below */ /* Note the
            use of xlocal[i] for &xlocal[i][0] */ if (rank < size - 1) for(i=0;i<maxn;i++){
                  MPI_Send(&xlocal[i][maxn / size], 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
            }
            if (rank > 0)
            for(i=0;i<maxn;i++){
                  MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0,
                              MPI_COMM_WORLD, &status);
            }/* Send down unless I'm at the bottom */ if (rank
            > 0)
            for(i=0;i<maxn;i++){
                  MPI_Send(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 1,
                              MPI_COMM_WORLD);
            }
            if (rank < size - 1) for(i=0;i<maxn;i++){
                  MPI_Recv(&xlocal[i][maxn / size + 1], 1, MPI_DOUBLE, rank + 1, 1,
            MPI_COMM_WORLD, &status); }/* Compute new values (but not on boundary) */
            itcnt++; diffnorm = 0.0; for (i = 1; i < maxn-1; i++) for (j = j_left; j <= j_right; j++)
                  { xnew[i][j] = (xlocal[i][j + 1] + xlocal[i][j - 1] +
                                    xlocal[i + 1][j] + xlocal[i - 1][j]) /
```

```
                                        4.0; diffnorm += (xnew[i][j] -
                    xlocal[i][j]) *
                                                (xnew[i][j] - xlocal[i][j]);
                }
        /* Only transfer the interior points */ for (i = 1;
        i < maxn-1; i++) for (j = j_left; j <= j_right; j++)
        xlocal[i][j] = xnew[i][j];
        MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                        MPI_COMM_WORLD);
        gdiffnorm = sqrt(gdiffnorm); if (rank == 0) printf("At iteration
        %d, diff is %e\n", itcnt, gdiffnorm);
    } while (gdiffnorm > 1.0e-2 && itcnt < 100); FILE *fp;
    char filename[80]; sprintf(filename,
    "Bprocessor%d.txt", rank); if ((fp = fopen(filename,
    "wb")) == NULL)
    { printf("Cannot open file.\n");
    }
    for (i = 1; i < maxn-1; i++)
    { for (j = j_left; j <= j_right; j++)
        { fprintf(fp, "%f ", xlocal[i][j]);
        } fprintf(fp, "\n");
    } fclose(fp);
    MPI_Finalize();
    return 0;
}
```

Commands to compile and run:

```
$ mpicc -o column_wise column_wise.c
$ mpirun -np 16 ./column_wise
```

APPENDIX F. MPI JACOBI ITERATIVE METHOD ON LAPLACE EQUATION GRID WISE WITH 4 PROCESSES

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
/* This example handles a 12 x 12 mesh, on 4 processors only. */
#define maxn 12 int main(argc,
argv) int argc; char **argv;
{ int rank, value, size, errcnt, toterr, i, j, itcnt; int i_first, i_last, j_left,
      j_right,i_cpu,j_cpu; MPI_Status status; double diffnorm,
      gdiffnorm;
      double xlocal[8][8]; double
      xnew[8][8];
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      if (size != 4)
            MPI_Abort(MPI_COMM_WORLD, 1);
      /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */ /* Note that top
      and bottom processes have one less row of interior points */

      i_cpu = 2; j_cpu
      = 2;

      i_first = 1; j_left = 1; i_last
      = maxn / i_cpu; j_right =
      maxn / j_cpu;

      for (i = 1; i < maxn / i_cpu; i++)
            for (j = 1; j < maxn / j_cpu; j++) xlocal[i][j] = rank;

      for (i = 0; i <= maxn/i_cpu; i++)
      { xlocal[0][i] = -1; xlocal[i][0]
            = -1; xlocal[6][i+1] = -1;
            xlocal[i+1][6] = -1;
      }

      itcnt = 0;
      do {
            /* Send up unless I'm at the top, then receive from below */ /* Note the
            use of xlocal[i] for &xlocal[i][0] */ if (rank == 0)
                        MPI_Send(xlocal[maxn/i_cpu], maxn/i_cpu, MPI_DOUBLE, 2, 0,
                           MPI_COMM_WORLD);
            if (rank == 0)
                  MPI_Recv(xlocal[0], maxn/i_cpu, MPI_DOUBLE, 2, 0,
                           MPI_COMM_WORLD, &status);
            if (rank == 2)
                  MPI_Send(xlocal[0], maxn/i_cpu, MPI_DOUBLE, 0, 0,
                           MPI_COMM_WORLD);
```

```
if (rank == 2)
	MPI_Recv(xlocal[maxn/i_cpu], maxn/i_cpu, MPI_DOUBLE, 0, 0,
			MPI_COMM_WORLD, &status);
if (rank == 1)
	MPI_Send(xlocal[maxn/i_cpu], maxn/i_cpu, MPI_DOUBLE, 3, 0,
			MPI_COMM_WORLD);
if (rank == 1)
	MPI_Recv(xlocal[0], maxn/i_cpu, MPI_DOUBLE, 3, 0,
			MPI_COMM_WORLD, &status);
if (rank == 3)
	MPI_Send(xlocal[0], maxn/i_cpu, MPI_DOUBLE, 1, 0,
			MPI_COMM_WORLD);
if (rank == 3)
	MPI_Recv(xlocal[maxn/i_cpu], maxn/i_cpu, MPI_DOUBLE, 1, 0,
			MPI_COMM_WORLD, &status);
/* Send up unless I'm at the top, then receive from below */ /* Note the
use of xlocal[i] for &xlocal[i][0] */ if (rank == 0)
	for (i = 1; i < (maxn / j_cpu); i++)
	{
		MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, 1, 0,
				MPI_COMM_WOR
	LD); }
if (rank == 0)
	for (i = 1; i < (maxn / j_cpu); i++)
	{
		MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, 1, 0,
				MPI_COMM_WORLD, &status);
	}
if (rank == 1)
	for (i = 1; i < (maxn / j_cpu); i++)
	{
		MPI_Send(&xlocal[i][0], 1, MPI_DOUBLE, 0, 0,
				MPI_COMM_WOR
	LD); }
if (rank == 1)
	for (i = 1; i < (maxn / j_cpu); i++)
	{
		MPI_Recv(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, 0, 0,
				MPI_COMM_WORLD, &status);
	}
if (rank == 2)
	for (i = 1; i < (maxn / j_cpu); i++)
	{
		MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, 3, 0,
				MPI_COMM_WORLD);
	}
if (rank == 2)
	for (i = 1; i < (maxn / j_cpu); i++)
	{
		MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, 3, 0,
```

```
                                MPI_COMM_WORLD, &status);
                }
        if (rank == 3)
                for (i = 1; i < (maxn / j_cpu); i++)
                {
                        MPI_Send(&xlocal[i][0], 1, MPI_DOUBLE, 2, 0,
                                        MPI_COMM_WOR
                LD); }
        if (rank == 3)
                for (i = 1; i < (maxn / j_cpu); i++)
                {
                                MPI_Recv(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, 2, 0,
                                 MPI_COMM_WORLD, &status);
                }/* Compute new values (but not on boundary) */ itcnt++;
        diffnorm = 0.0; for (i = i_first; i < i_last; i++) for (j = j_left; j <
        j_right; j++)
                { xnew[i][j] = (xlocal[i][j + 1] + xlocal[i][j - 1] + xlocal[i + 1][j] + xlocal[i -
                        1][j]) /
                                        4.0; diffnorm += (xnew[i][j] -
                        xlocal[i][j]) *
                                                (xnew[i][j] - xlocal[i][j]);
                }
        /* Only transfer the interior points */ for (i =
        i_first; i < i_last; i++) for (j = j_left; j < j_right;
        j++) xlocal[i][j] = xnew[i][j];
        MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                                MPI_COMM_WORLD); gdiffnorm =
        sqrt(gdiffnorm); if (rank == 0) printf("At iteration %d, diff is
        %e\n", itcnt, gdiffnorm);
    } while (gdiffnorm > 1.0e-4 && itcnt < 200); FILE *fp;
    char filename[80]; sprintf(filename,
    "ABprocessor%d.txt", rank); if ((fp = fopen(filename,
    "wb")) == NULL)
    { printf("Cannot open file.\n");
    } for (i = i_first; i < i_last; i++)
    { for (j = j_left; j < j_right; j++)
        { fprintf(fp, "%f ", xlocal[i][j]);
        } fprintf(fp, "\n");
    } fclose(fp);
    MPI_Finalize();
    return 0;
}

Commands to compile and run:
$ mpicc -o both2x2 both2x2.c
$ mpirun -np 4 ./both2x2
```

APPENDIX G. MPI JACOBI ITERATIVE METHOD ON LAPLACE EQUATION GRID WISE WITH 16 PROCESSES IN 4X4 GRID

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
/* This example handles a 12 x 12 mesh, on 16 processors only. */
#define maxn 12 int main(argc,
argv) int argc; char **argv;
{ int rank, value, size, errcnt, toterr, i, j, itcnt,l,m,p; int i_first, i_last, j_left,
      j_right, i_cpu, j_cpu; MPI_Status status; double diffnorm, gdiffnorm;
      double xlocal[6][6]; double xnew[6][6];
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      if (size != 16)
            MPI_Abort(MPI_COMM_WORLD, 1);
      /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */
      /* Note that top and bottom processes have one less row of interior points */

      i_cpu = 4; j_cpu =
      4; i_first = 1;
      j_left = 1; i_last =
      i_cpu; j_right =
      j_cpu;



      if (rank < i_cpu ){ i_first++;
           } if (rank >= size-i_cpu ){
             i_last--;
           }

      if (rank%i_cpu == 0){ j_left++;}
      if (rank%i_cpu == 3){
            j_right--;
           }


      for (i = i_first; i < i_last; i++)
                              for (j = j_left; j < j_right; j++)
                  xlocal[i][j] = 6;


      for (j = 0; j <= i_cpu; j++)
      { xlocal[i_first - 1][j] = -1;
            xlocal[i_last][j] = -1;
      }

      for (i = 0; i <= j_cpu; i++)
```

```
{ xlocal[i][j_left -1 ] = -1;
        xlocal[i][j_right] = -1;
}


printf("\nrank = %d\n ", rank); for (i = 0; i
< 5; i++)
{
        for (j = 0; j < 5; j++)
        { printf("%f ", xlocal[i][j]);
        } printf("\n");
}

itcnt = 0;
do {
        /* Send up unless I'm at the top, then receive from below */ /* Note the use of xlocal[i] for &xlocal[i][0]
        */ if(rank >= i_cpu && rank <= size-i_cpu-1 && rank%i_cpu != 0 && rank%i_cpu != i_cpu for (i = 1; i <=
        maxn / i_cpu; i++){
                MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                        &status);
                MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i][maxn/j_cpu+1], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD, &status);}
            for (j = 1; j <= maxn / j_cpu; j++){
                MPI_Send(&xlocal[maxn/j_cpu][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD);
                MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD,
                        &status);
                MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[maxn/j_cpu+1][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD,
                        &status);}
        }

        if(rank < i_cpu-1 && rank != 0){ for (i = 2; i <=
            maxn / i_cpu; i++){
                MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                        &status);
                MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i][maxn/j_cpu+1], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD, &status);}
            for (j = 1; j <= maxn / j_cpu; j++){
                MPI_Send(&xlocal[maxn/j_cpu][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[maxn/j_cpu+1][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD,
                        &status);}
```

```
}

if(rank > size-i_cpu && rank != size-1){ for (i = 1; i <
      maxn / i_cpu; i++){
            MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, rank + 1, 0,
                  MPI_COMM_WORLD);
            MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                  &status);
            MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                  MPI_COMM_WORLD);
            MPI_Recv(&xlocal[i][maxn/j_cpu+1], 1, MPI_DOUBLE, rank + 1, 0,
                  MPI_COMM_WORLD, &status);}
      for (j = 1; j <= maxn / j_cpu; j++){
            MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD,
                  &status);
            MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD);}
}

if(rank%i_cpu == 0 && rank != 0 && rank != size-i_cpu ){ for (i = 1; i <=
      maxn / i_cpu; i++){
            MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, rank + 1, 0,
                  MPI_COMM_WORLD);
            MPI_Recv(&xlocal[i][maxn/j_cpu+1], 1, MPI_DOUBLE, rank + 1, 0,
                  MPI_COMM_WORLD, &status);}
      for (j = 2; j <= maxn / j_cpu; j++){
            MPI_Send(&xlocal[maxn/j_cpu][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                  MPI_COMM_WORLD);
            MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD,
                  &status);
            MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                  MPI_COMM_WORLD);
            MPI_Recv(&xlocal[maxn/j_cpu+1][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD,
                  &status);}
}

if(rank%i_cpu == i_cpu-1 && rank != i_cpu-1 && rank != size-1){ for (i = 1; i <=
      maxn / i_cpu; i++){
            MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                  &status);
            MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                  MPI_COMM_WORLD);}
      for (j = 1; j < maxn / j_cpu; j++){
            MPI_Send(&xlocal[maxn/j_cpu][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD);
            MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD,
                  &status);
            MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                  MPI_COMM_WORLD);
            MPI_Recv(&xlocal[maxn/j_cpu+1][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD,
                  &status);}
}
```

```
if(rank == 0){ for (i = 2; i <= maxn / i_cpu; i++){
            MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD);
            MPI_Recv(&xlocal[i][maxn/j_cpu+1], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD, &status);}
      for (j = 2; j <= maxn / j_cpu; j++){
            MPI_Send(&xlocal[maxn/j_cpu][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                        MPI_COMM_WORLD);
            MPI_Recv(&xlocal[maxn/j_cpu+1][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD,
                        &status);}
}

if(rank == i_cpu-1){ for (i = 2; i <= maxn / i_cpu;
      i++){
            MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                        &status);
            MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD);}
      for (j = 1; j < maxn / j_cpu; j++){
            MPI_Send(&xlocal[maxn/j_cpu][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                        MPI_COMM_WORLD);
            MPI_Recv(&xlocal[maxn/j_cpu+1][j], 1, MPI_DOUBLE, rank + j_cpu, 0, MPI_COMM_WORLD,
                        &status);}
}

if(rank == size-i_cpu){ for (i = 1; i < maxn /
      i_cpu; i++){
            MPI_Send(&xlocal[i][maxn/j_cpu], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD);
            MPI_Recv(&xlocal[i][maxn/j_cpu+1], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD, &status);}
      for (j = 2; j <= maxn / j_cpu; j++){
            MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD,
                        &status);
            MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD);}
}

if(rank == size-1){
      for (i = 1; i < maxn / i_cpu; i++){
            MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                        &status);
            MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD);}
      for (j = 1; j < maxn / j_cpu; j++){
            MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD,
                        &status);
            MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0, MPI_COMM_WORLD);}
}
```

```
        /* Compute new values (but not on boundary) */
            itcnt++; diffnorm = 0.0; for (i = i_first; i < i_last; i++)
            for (j = j_left; j < j_right; j++)
                { xnew[i][j] = (xlocal[i][j + 1] + xlocal[i][j - 1] +
                                                    xlocal[i + 1][j] + xlocal[i - 1][j]) /
                                    4.0; diffnorm += (xnew[i][j] -
                    xlocal[i][j]) *
                                        (xnew[i][j] - xlocal[i][j]);
                }
            /* Only transfer the interior points */ for (i =
            i_first; i < i_last; i++) for (j = j_left; j < j_right;
            j++) xlocal[i][j] = xnew[i][j];
            MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                            MPI_COMM_WORLD); gdiffnorm =
            sqrt(gdiffnorm); if (rank == 0) printf("At iteration %d, diff is
            %e\n", itcnt, gdiffnorm);
        } while (gdiffnorm > 1.0e-2 && itcnt < 100); FILE *fp;
        char filename[80]; sprintf(filename,
        "ABprocessor%d.txt", rank); if ((fp = fopen(filename,
        "wb")) == NULL)
        { printf("Cannot open file.\n");
        } for (i = i_first; i < i_last; i++)
        {
            for (j = j_left; j < j_right; j++)
            { fprintf(fp, "%f ", xlocal[i][j]);
            } fprintf(fp, "\n");
        } fclose(fp);
          MPI_Finalize(); return 0;
}
```

Commands to compile and run:

```
$ mpicc -o both4x4 both4x4.c
$ mpirun -np 16 ./both4x4
```

APPENDIX H. MPI JACOBI ITERATIVE METHOD ON LAPLACE EQUATION GRID WISE WITH 16 PROCESSES IN 2X16 GRID

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
/* This example handles a 32 x 32 mesh, on 16 processors only. */
#define maxn 32 int main(argc,
argv) int argc; char **argv;
{ int rank, value, size, errcnt, toterr, i, j, itcnt, l, m, p; int i_first, i_last, j_left, j_right,
        i_cpu, j_cpu; MPI_Status status; double diffnorm, gdiffnorm; double
        xlocal[18][6]; double xnew[18][6];
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size != 16)
                MPI_Abort(MPI_COMM_WORLD, 1);
        /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */
        /* Note that top and bottom processes have one less row of interior points */
        i_cpu = 2; j_cpu
        = 8; i_first = 1;
        j_left = 1; i_last
        = 16; j_right = 4;

        if (rank < j_cpu)
        { i_first++;
        } if (rank >= size - j_cpu)
        { i_last--;
        }
        if (rank % j_cpu == 0)
        { j_left++;
        } if (rank % j_cpu == j_cpu-1)
        { j_right--;
        }


    for (i = i_first; i <= i_last; i++)
        for (j = j_left; j <= j_right; j++)
            xlocal[i][j] = 6;


    for (j = 0; j <= j_right+1; j++)
    { xlocal[i_first - 1][j] = -1;
        xlocal[i_last+1][j] = -1;
    }

    for (i = 0; i <= i_last+1; i++)
    { xlocal[i][j_left-1] = -1;
        xlocal[i][j_right+1] = -1;
    }
```

```
printf("\nrank = %d\n ", rank); for (i = 0; i
< 18; i++)
{
        for (j = 0; j < 6; j++)
        { printf("%f ", xlocal[i][j]);
        } printf("\n");
}

itcnt = 0;
do {
        /* Send up unless I'm at the top, then receive from below */ /* Note the use of xlocal[i] for
        &xlocal[i][0] */ if (rank >= j_cpu && rank <= size - j_cpu - 1 && rank % j_cpu != 0 && rank %
        {
                for (i = 1; i <= i_last; i++)
                {
                        MPI_Send(&xlocal[i][j_right], 1, MPI_DOUBLE, rank + 1, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0,
                                MPI_COMM_WORLD, &status);
                        MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i][j_right + 1], 1, MPI_DOUBLE, rank + 1, 0
                                MPI_COMM_WORLD, &status);
                }
                for (j = 1; j <= j_right; j++)
                {
                        MPI_Send(&xlocal[i_last][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                                MPI_COMM_WORLD, &status);
                        MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i_last + 1][j], 1, MPI_DOUBLE, rank + j_cpu
                MPI_COMM_WORLD, &status); }
        }

        if (rank < j_cpu - 1 && rank != 0)
        {
                for (i = 2; i <= i_last; i++)
                {
                        MPI_Send(&xlocal[i][j_right], 1, MPI_DOUBLE, rank + 1, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0,
                                MPI_COMM_WORLD, &status);
                        MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i][j_right + 1], 1, MPI_DOUBLE, rank + 1, 0
                                MPI_COMM_WORLD, &status);
                }
                for (j = 1; j <= j_right; j++)
                {
```

```
                        MPI_Send(&xlocal[i_last][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i_last + 1][j], 1, MPI_DOUBLE, rank + j_cpu
                                MPI_COMM_WORLD, &status);
                }
        }

        if (rank > size - j_cpu && rank != size - 1)
        {
                for (i = 1; i < i_last; i++)
                {
                        MPI_Send(&xlocal[i][j_right], 1, MPI_DOUBLE, rank + 1, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0,
                                MPI_COMM_WORLD, &status);
                        MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i][j_right + 1], 1, MPI_DOUBLE, rank + 1, 0
                                MPI_COMM_WORLD, &status);
                }
                for (j = 1; j <= j_right; j++)
                {
                        MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                                MPI_COMM_WORLD, &status);
                        MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                                MPI_COMM_WORLD);
                }
        }

        if (rank % j_cpu == 0 && rank != 0 && rank != size - j_cpu)
        {
                for (i = 1; i <= i_last; i++)
                {
                        MPI_Send(&xlocal[i][j_right], 1, MPI_DOUBLE, rank + 1, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i][j_right + 1], 1, MPI_DOUBLE, rank + 1, 0
                                MPI_COMM_WORLD, &status);
                }
                for (j = 2; j <= j_right; j++)
                {
                        MPI_Send(&xlocal[i_last][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                                MPI_COMM_WORLD, &status);
                        MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                                MPI_COMM_WORLD);
                        MPI_Recv(&xlocal[i_last + 1][j], 1, MPI_DOUBLE, rank + j_cpu
                                MPI_COMM_WORLD, &status);
                }
        }
```

```
if (rank % j_cpu == j_cpu - 1 && rank != j_cpu - 1 && rank != size - 1)
{
        for (i = 1; i <= i_last; i++)
        {
                MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD, &status);
                MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD);
        }
        for (j = 1; j < j_right; j++)
        {
                MPI_Send(&xlocal[i_last][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD, &status);
                MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i_last + 1][j], 1, MPI_DOUBLE, rank + j_cpu
                        MPI_COMM_WORLD, &status);
        }
}

if (rank == 0)
{
        for (i = 2; i <= i_last; i++)
        {
                MPI_Send(&xlocal[i][j_right], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i][j_right + 1], 1, MPI_DOUBLE, rank + 1, 0
                        MPI_COMM_WORLD, &status);
        }
        for (j = 2; j <= j_right; j++)
        {
                MPI_Send(&xlocal[i_last][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i_last+ 1][j], 1, MPI_DOUBLE, rank + j_cpu,
                        MPI_COMM_WORLD, &status);
        }
}

if (rank == j_cpu - 1)
{
        for (i = 2; i <= i_last; i++)
        {
                MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD, &status);
                MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD);
        }
        for (j = 1; j < j_right; j++)
```

```
        {
                MPI_Send(&xlocal[i_last][j], 1, MPI_DOUBLE, rank + j_cpu, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i_last + 1][j], 1, MPI_DOUBLE, rank + j_cpu
                        MPI_COMM_WORLD, &status);
        }
}

if (rank == size - j_cpu)
{
        for (i = 1; i < i_last; i++)
        {
                MPI_Send(&xlocal[i][j_right], 1, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD);
                MPI_Recv(&xlocal[i][j_right + 1], 1, MPI_DOUBLE, rank + 1, 0
                        MPI_COMM_WORLD, &status);
        }
        for (j = 2; j <= j_right; j++)
        {
                MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD, &status);
                MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD);
        }
}

if (rank == size - 1)
{
        for (i = 1; i < i_last; i++)
        {
                MPI_Recv(&xlocal[i][0], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD, &status);
                MPI_Send(&xlocal[i][1], 1, MPI_DOUBLE, rank - 1, 0,
                        MPI_COMM_WORLD);
        }
        for (j = 1; j < j_right; j++)
        {
                MPI_Recv(&xlocal[0][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD, &status);
                MPI_Send(&xlocal[1][j], 1, MPI_DOUBLE, rank - j_cpu, 0,
                        MPI_COMM_WORLD);
        }
}
/* Compute new values (but not on boundary) */
itcnt++; diffnorm = 0.0; for (i = i_first; i <= i_last; i++) for
(j = j_left; j <= j_right; j++)
        { xnew[i][j] = (xlocal[i][j + 1] + xlocal[i][j - 1] +
                                                xlocal[i + 1][j] + xlocal[i - 1][j]) /
                                        4.0;
                diffnorm += (xnew[i][j] - xlocal[i][j]) *
                                (xnew[i][j] - xlocal[i][j]); }
```

```
                    /* Only transfer the interior points */ for (i = i_first; i
                    <= i_last; i++) for (j = j_left; j <= j_right; j++)
                    xlocal[i][j] = xnew[i][j];
                    MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                                        MPI_COMM_WORLD);
                    gdiffnorm = sqrt(gdiffnorm); if (rank == 0) printf("At iteration %d,
                    diff is %e\n", itcnt, gdiffnorm);
          } while (gdiffnorm > 1.0e-10 && itcnt < 10000); FILE *fp;
          char filename[80]; sprintf(filename, "ABprocessor%d.txt",
          rank); if ((fp = fopen(filename, "wb")) == NULL)
          { printf("Cannot open file.\n");
          } for (i = i_first; i <= i_last; i++)
          {
                    for (j = j_left; j <= j_right; j++)
                    { fprintf(fp, "%f ", xlocal[i][j]);
                    } fprintf(fp, "\n");
          } fclose(fp);
          MPI_Finalize();
          return 0;
}
```

Commands to compile and run:

```
$ mpicc -o both2x16 both2x16.c
$ mpirun -np 16 ./both2x16
```

APPENDIX I. JACOBI

```
a = [4 2 3; 3 -5 2; -2 3 8]; b= [8 -14 27]; x1= 0;x2=0;x3=0; for
i=1:100 x1_new = (b(1,1) - a(1,2)*x2 - a(1,3)*x3)/a(1,1);
x2_new = (b(1,2) - a(2,1)*x1 - a(2,3)*x3)/a(2,2); x3_new =
(b(1,3) - a(3,1)*x1 - a(3,2)*x2)/a(3,3); x1=x1_new;
x2=x2_new; x3=x3_new; end
        p= [x1 x2 x3]
```

APPENDIX J. JACOBI NTH ORDER

```
function out = jacob1(A, B,iter)
A=rand(10,10); B=rand(10,1);
N=length(A);
AA=A;

for i=1:N
      AA(i,i) = 0;
      DAA(i,1)=A(i,i);

end Xint = zeros(N,1);

for i=1:iter

      out= (B'-AA*Xint)./DAA;
      Xint=out;

end
out
```

APPENDIX K. 1D POISON EQUATION

```
function f = oneDpoison(h,x_int,x_fin,y_int,y_fin)

h=0.1;x_int=0;x_fin=pi;y_int=0;y_fin=0; N =
ceil((x_fin-x_int)/h);

for i=1:N-1 x(i)
= i*h; end

u = zeros(N-1,N-1);
u(1,1) = -2; u(1,2) = 1;
for i=2:N-2
u(i,i-1) = 1; u(i,i) = -2;
u(i,i+1) = 1;
end u(end, end-1)
=1; u(end, end)= -
2; for i=1:N-1
        y(i) = (h^2)*sin(x(i));
end


y(1) = y(1)-(y_int); y(end) = y(end)-(y_fin);%solution is known at this
node f= inv(u)*y';

plot(x',f,'ok'); hold on
plot(x,-sin(x),'-r');
```

APPENDIX L. 2D POISON EQUATION

```
clear all;
N=5;M=N; h =
pi/(N-1);

s2 = 4*ones(M,1); s =
-1*ones(M-1,1);
A = diag(s2,0) + diag(s,1) + diag(s,-1);

A = sparse(A);

I = speye(M);
B = kron(A,I) + kron(I,A); B=

full(B);

for i=1:N
    for j=1:N
        x(i) = (i-1)*h; y(j) = (j-1)*h; f(i,j) = -
        2.*sin(x(i)).*sin(y(j)).*h^2; index = (i-1)*N + j;
        der(index) = f(i,j);
    end
end

U = -(B/der);
%u_exact solution

for i=1:N
    for j=1:N
        x(i) = (i-1)*h; y(j) = (j-1)*h; P(i,j) =
    sin(x(i)).*sin(y(j)); index = (i-1)*N + j;
    u_e(index) = P(i,j);

    end
end u_e;
u_e = u_e';


plot(U, '- g'); hold
on plot(u_e, 'o r');
```