APPENDIX A

Vector and Matrix Norms

The norm of a vector v is a positive real number which gives some measure of the size of the vector and is denoted by $\|v\|$. The norm must satisfy the following axioms:

(i) $\|v\| > 0$ if $v \neq 0$ and $\|v\| = 0$ iff $v = 0$

(ii) $\|cv\| = |c|\|v\|$ for a real or complex scalar c.

(iii) $\|v + w\| \leq \|v\| + \|w\|$ (the triangular inequality)

If the $n \times 1$ vector v has components $v_1, v_2, \cdots v_n$, then the three most commonly used norms are defined as follows:

The 1-norm of v is the sum of the moduli of the components of v, that is

$$\|v\| = |v_1| + |v_2| + \cdots + |v_n| = \sum_{i=1}^{n} |v_1|$$

the infinity norm of v is the maximum of the moduli of the components of v, that is:

$$\|v\|_{\infty} = \underset{i=1}{\overset{n}{Max}} |v_i|$$

the 2-norm of v is the square root of the sum of the squares of the moduli of the components of v, that is:

$$\|v\|_2 = \left(|v_1|^2 + |v_2|^2 + \cdots + |v_n|^2\right)^2 = \left[\sum_{i=1}^{n} |v_i|^2\right].$$

Thus, the 2-norm is just the length of the vector. The norm of matrix A is a real positive number giving a measure of the size of a matrix and must satisfy the following axioms:

(i) $\|A\| > 0$ if $A \neq 0$ and $\|A\| = 0$ if $A = 0$

(ii) $\|cA\| = |c|\|cA\|$ for a real or complex scalar c.

(iii) $\|A + B\| \leq \|A\| + \|B\|$ (the triangular inequality)

(iv) $\|AB\| \leq \|A\|\|B\|$ (the Scharz inequality)

vectors and matrices occur together so it is essential that they satisfy a condition equivalent to (iv). As a consequence, matrix and vector norms are said to be compatible or consistent if :

$$\|Av\| \leq \|A\|\|v\|, \quad v \neq 0$$

let A be an $n \times n$ matrix and v a member of the set S of $n \times 1$ vectors whose norms are unity, that is , $v \in S$ if $\|v\| = 1$. In general, the norm of the vector $\|Av\|$ will vary as $\|v\|$ varies. Let $\|v_o\|$ be a member of S that makes $\|Av\|$ attain its maximum value. Then the norm of the matrix $\|A\|$ is defined by:

$$\|A\| = \|Av_o\| = \underset{\|v\|=1}{Max}\|Av\|$$

this matrix norm is said to be subordinate to the vector norm and automatically compatibility condition, because if $v = v_1$ is any member of S ,

$$\|Av_1\| \leq \|Av_o\| = \|A\| = \|A\|\|v_1\|,$$

since $\|v_1\| = 1$. It follows that for all subordinate matrix norms,

$$\|I\| = \underset{\|v\|=1}{Max}\|Iv\| = \underset{\|v\|=1}{Max}\|v\|,$$

where I is the unit matrix. The definitions of the Eq. (3.11) and ∞ norms with $\|v\| = 1$ lead to the following results: (1) the 1-norm of the matrix A is the maximum column sum of the moduli of elements of A, (2) the ∞-norm of the matrix A is the maximum row sum of the moduli of elements of A, (3) the 2-norm of the matrix A is the square root of the spectra radius of $A^T A$, where A^T is the transpose of the conjugate complex

of A and (4) the spectra radius of a matrix B written $\rho(B)$ and defined as the maximum of the moduli of its eigenvalues $\lambda_i (i = 1, \cdots, n)$. Recall that eigenvalues are calculated from $\det(B - \lambda_i) = 0$. We will often meet real symmetric matrices for which $A^T = A$. If x is an eigenvector of the matrix A corresponding to the eigenvalue λ then $Ax = \lambda^2 x$. Hence,

$$A^2 x = A(Ax) = A^2 x = \lambda Ax = \lambda^2 x$$

and thus,

$$\|A\|_2 = \sqrt{\rho(A^2)} = \rho(A) = \underset{i}{Max}|\lambda_i| \qquad [A.1]$$

we often encounter tridiagonal matrices. The eigenvalues of the $n \times n$ tridiagonal matrix are:

$$\lambda_s = a + 2\sqrt{bc}\cos(s\pi / n + 1), \quad s = 1, \cdots, n \qquad [A.2]$$

where a, b and c may be real or complex. Take the positive root; let λ_i be an eigenvalue of the $n \times n$ matrix B and x_i the corresponding eigenvector. Then we have

$Bx_i = \lambda x_i$ and so

$$\|Bx_i\| = \|\lambda x_i\| = |\lambda_i| \|x_i\|$$

thus for all compatible matrix and vector norms it follows that

$$|\lambda_i| \|x_i\| = \|Bx_i\| \le \|B\| \|x_i\|$$

and so

$|\lambda_i| \le \|B\|$ $\quad i = 1, \cdots, n$. Therefore,

$$\rho(B) \le \|B\| \qquad [A.3]$$

thus if $\rho(B) > 1$ then $\|B\| > 1$.

APPENDIX B

MPI Program for 1-D Explicit Parabolic

```
/******** MPI-1D *********/
#include "mpi.h"
#include <stdio.h>
#define ARRAYSIZE
#define MASTER    0
#define MAXWORKER
#define STEPS
#define NGHBOR1
#define NGHBOR2
int main(int argc, char *argv[])
{
        void prtdat();
        int numtasks, numworkers, taskid, dest, rc, i, ix, it, index, rows, start, end,
        extra, left, right, msgtag, arraymsg = 1, indexmsg = 2, source, chunksize,
        arraysize;
        float data[ARRAYSIZE];
        float result[ARRAYSIZE];
        float result1[ARRAYSIZE];
        FILE *fp;
```

```
MPI_Status status;
/*********** Initializations ******************************/
rc = MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
if (taskid != 0)
        printf ("error initializing MPI and obtaining task ID information\n");
else
        printf ("mpi_heat1D MPI task ID = %d\n", taskid);
printf("%d tasks, I am task %d\n", numtasks, taskid);
numworkers = numtasks - 1;
chunksize = (ARRAYSIZE/numworkers);
/*****************************Master task ********************/
if (taskid == MASTER)
{
        printf("\n**********************Starting MPI Example 1 *****\n");
        printf("MASTER: number of worker tasks will be= %d\n",
        numworkers);
        MPI_Finalize();
      /* Initialize the array */
      for (ix = 0; ix < ARRAYSIZE; ix++)
                data[ix] = 0.0;
      index = 0;
      /* Send each worker task its portion of the array */
      chunksize = ARRAYSIZE/numworkers;
      extra = ARRAYSIZE%numworkers;
```

```
index = 0;
for (i = 1; i <= numworkers; i++)
   {
          rows = (i <= extra) ? chunksize+1 : chunksize;
        /* Tell each worker who its neighbor are */
        if (i == 1)
                left = 0;
          else
                left = i - 1;
          if (i == numworkers)
                right = 0;
          else
                right = i + 1;
        /* Now send startup information to each worker */
        dest = i;
          msgtag = arraymsg;
       MPI_Send(&index, 1, MPI_INT, dest, msgtag,
       MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, dest, msgtag,
       MPI_COMM_WORLD);
        MPI_Send(&left, 1, MPI_INT, dest, msgtag,
       MPI_COMM_WORLD);
        MPI_Send(&right, 1, MPI_INT, dest, msgtag,
       MPI_COMM_WORLD);
        MPI_Send(&data[index], rows, MPI_FLOAT, dest, msgtag,
       MPI_COMM_WORLD);
```

```
        printf("Sent to= %d index= %d rows= %d left= %d right=
       %d\n",dest,index,rows,left,right);
        index = index + rows;
    }
    /* Now wait for result from all worker task */
    for (i = 1; i < numworkers; i++)
    {
        source = i;
        msgtag = indexmsg;
        MPI_Recv(&index, 1, MPI_INT, source, msgtag,
        MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, source, msgtag,
        MPI_COMM_WORLD, &status);
        MPI_Recv(&result[index], rows, MPI_FLOAT, source, msgtag,
        MPI_COMM_WORLD, &status);
    }
    /* write final output */
    printf(".....\n");
    printf("MASTER: Sample results from worker task = %d\n", source);
     printf(" result1[%d]=%f\n", index, result1[index]);
    printf(" result1[%d]=%f\n", index+100, result1[index+100]);
    printf(" result1[%d]=%f\n\n", index+1000, result1[index+1000]);
    MPI_Finalize();
    }
    }
/*****************************Worker task ************************/
```

```
if (taskid != MASTER)
{
        /* Receive my portion of array from the master task */
        source = MASTER;
        msgtag = arraysize;
        MPI_Recv(&index, 1, MPI_INT, source, msgtag,
        MPI_COMM_WORLD, &status);
         MPI_Recv(&rows, 1, MPI_INT, source, msgtag,
        MPI_COMM_WORLD, &status);
        MPI_Recv(&left, 1, MPI_INT, source, msgtag, MPI_COMM_WORLD,
        &status);
        MPI_Recv(&right, 1, MPI_INT, source, msgtag,
        MPI_COMM_WORLD, &status);
        MPI_Recv(&data[index], rows, MPI_FLOAT, source, msgtag,
        MPI_COMM_WORLD, &status);
        /* Copy my data into working array */
        result = malloc(sizeof(float) * (chunksize+2));
        result1 = malloc(sizeof(float) * (chunksize+2));
        if (index==0)
                start = 1;
        else
                start = index;
        if ((index+rows)==ARRAYSIZE)
                end = index + rows-2;
        else
                end = index + rows-1;
```

```
for (i = index; i < index + chunksize; i++)
        result[i] = data[i];
 /* Do the step iteration */
 for (it = 1; it <= STEPS; it++)
 {
        if (left != 0)
        {
        MPI_Send(&data[index], 1, rows, left, NGHBOR2,
        MPI_COMM_WORLD);
                source = left;
                msgtag = NGHBOR1;
                MPI_Recv(&data[index-1], 1, rows, source, msgtag,
                MPI_COMM_WORLD, &status);
        }
        if (right != 0)
        {
                MPI_Send(&data[index+rows-1], 1, rows, right,
                indexmsg, MPI_COMM_WORLD);
                source = right;
                msgtag = NGHBOR2;
                MPI_Recv(&data[index+rows], 1, rows, source, msgtag,
                MPI_COMM_WORLD, &status);
        }
        /* do the calculation */
        for (ix = 1; ix <= rows; ix++)
        {
```

```
                    result1[ix] = result[ix] + 0.5 * (result[ix-1] -
                                    2 * result[ix] + result[ix+1]);
                }
            }
            /* Send my results back to the master task */
            MPI_Send(&index, 1, MPI_INT, MASTER, indexmsg,
            MPI_COMM_WORLD);
            MPI_Send(&rows, 1, MPI_INT, MASTER, indexmsg,
            MPI_COMM_WORLD);
            MPI_Send(&result1[index], rows, MPI_FLOAT, MASTER, indexmsg,
            MPI_COMM_WORLD);
            }
            MPI_Finalize();
}
```

APPENDIX C

MPI Program for 2-D Explicit Parabolic Equation

```
/**************************************************************
 * Heat-2D
 **************************************************************/
#include "mpi.h"
#include <stdio.h>
#define NXPROB          /* x dimension of grid problem */
#define NYPROB          /* y dimension of grid problem */
#define STEPS           /* number of time steps       */
#define MAXWORKER       /* maximum number of worker task */
#define MINWORKER       /* minimum number of worker task */
#define BEGIN           /* message type   */
#define NGHBOR1         /* message type */
#define NGHBOR2         /* message type */
#define NONE            /* indicates no neighbor */
#define DONE            /* message type */
#define MASTER          /* taskid of first process */
struct Parms {
        float cx;
        float cy;
```

```
} parms = {0.1, 0.1};
main(int argc, char * argv[])
{
        void inidat(), prtdat(), update();
        float start_time, end_time;
      float u[2][NXPROB][NYPROB];
        int taskid, numworkers, numtasks, averow, rows,
          offset, extra, dest, source, neighbor1, neighbor2,
          msgtype, nbytes, rc, start, end,
          i, ix, iy, iz, it;
        MPI_Status status;
        /* First, find out my taskid and how many tasks are running */
        rc = MPI_Init(&argc, &argv);
        rc = MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        rc = MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
        if (rc != 0)
                printf ("error initializing MPI and obtaining task ID information\n");
        else
                printf ("mpi_heat2D MPI task ID = %d\n", taskid);
        numworkers = numtasks - 1;
        if (taskid == MASTER)
        {
                /***************master code*****************************/
                /* Check if numworkers is within range - quit if not */
                if ((numworkers > MAXWORKER) || (numworkers < MINWORKER))
                {
```

```
        printf("numworkers %d for this example\n", numworkers);
        printf("MP_PROCS needs to be between %d and %d for this
        exercise\n", MINWORKER, MAXWORKER);
        MPI_Finalize();
//      exit(-1);
}
/* Initialize grid */
printf("Grid size: X= %d Y= %d Time steps=
%d\n",NXPROB,NYPROB,STEPS);
printf("Initializing grid and writing initial.dat file...\n");
inidat(NXPROB, NYPROB, u);
start_time = MPI_Wtime();
prtdat(NXPROB, NYPROB, u, "initial.dat");
/* Distribute work to workers. Must first figure out how many rows to
send and what to do with extra rows */
averow = NXPROB/numworkers;
extra = NXPROB%numworkers;
offset = 0;
for (i = 1; i <= numworkers; i++)
{
        rows = (i <= extra) ? averow+1 : averow;
        /* Tell each worker who its neighbors are, since they must
        exchange data with each other. */
        if (i == 1)
                neighbor1 = NONE;
        else
```

```
            neighbor1 = i - 1;
        if (i == numworkers)
            neighbor2 = NONE;
        else
            neighbor2 = i + 1;
        /* Now send startup information to each worker */
        dest = i;
        MPI_Send(&offset, 1, MPI_INT, dest, BEGIN,
        MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, dest, BEGIN,
        MPI_COMM_WORLD);
        MPI_Send(&neighbor1, 1, MPI_INT, dest, BEGIN,
        MPI_COMM_WORLD);
        MPI_Send(&neighbor2, 1, MPI_INT, dest, BEGIN,
        MPI_COMM_WORLD);
        MPI_Send(&u[0][offset][0], rows*NYPROB, MPI_FLOAT,
        dest, BEGIN, MPI_COMM_WORLD);
        printf("Sent to= %d offset= %d rows= %d neighbor1= %d
        neighbor2= %d\n", dest, offset, rows, neighbor1, neighbor2);
        offset = offset + rows;
}
/* Now wait for results from all worker tasks */
for (i = 1; i <= numworkers; i++)
{
        source = i;
        msgtype = DONE;
```

```
                MPI_Recv(&offset, 1, MPI_INT, source, msgtype,
                MPI_COMM_WORLD, &status);
                MPI_Recv(&rows, 1, MPI_INT, source, msgtype,
                MPI_COMM_WORLD, &status);
                MPI_Recv(&u[0][offset][0], rows*NYPROB, MPI_FLOAT,
                source, msgtype, MPI_COMM_WORLD, &status);
        }
        /* Write final output and call X graph */
        printf("Writing final.dat file and gegerating graph...\n");
        prtdat(NXPROB, NYPROB, &u[0][0][0], "final.dat");
        end_time = MPI_Wtime();
        printf("using %d processors and %f seconds \n\n",
                        numworkers, end_time - start_time);
        MPI_Finalize();
}  /* End of master code */
if (taskid != MASTER)
{
        /********* Worker code *********************************/
        /* Initialize everything - including the borders - to zero */
        for (iz = 0; iz < 2; iz++)
        {
                for ( ix = 0; ix < NXPROB; ix++)
                {
                        for (iy = 0; iy < NYPROB; iy++)
                        {
                                u[iz][ix][iy];
```

```
                    }
                }
            }

    /* Now receive my offset, rows, neighbors, and grid partition from master
    */
    source = MASTER;
    msgtype = BEGIN;
MPI_Recv(&offset, 1, MPI_INT, source, msgtype,
MPI_COMM_WORLD, &status);
MPI_Recv(&rows, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD,
&status);
MPI_Recv(&neighbor1, 1, MPI_INT, source, msgtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&neighbor2, 1, MPI_INT, source, msgtype,
MPI_COMM_WORLD, &status);
    MPI_Recv(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, source,
msgtype, MPI_COMM_WORLD, &status);
    /* Determine border elements. Need to consider first and last columns.
Obviously, row 0 can't exchange with row 0-1. likewise, the last row can't
exchange with last+1. */
    if (offset==0)
        start = 1;
    else
        start = offset;
    if ((offset+rows)==NXPROB)
```

```
        end = offset + rows-2;
    else
        end = offset + rows-1;

    /* Begin doing step iterations. Must communicate border elements with
    neighbors. if i have the first or last grid row, then i only need to
    communicate with one neighbor */
    iz = 0;
    for ( it = 1; it <= STEPS; it++)
    {
        if (neighbor1 != NONE)
        {
                MPI_Send(&u[iz][offset][0], NYPROB, MPI_FLOAT,
       neighbor1, NGHBOR2, MPI_COMM_WORLD);
                source = neighbor1;
                msgtype = NGHBOR1;
               MPI_Recv(&u[iz][offset-1][0], NYPROB,
               MPI_FLOAT, source, msgtype, MPI_COMM_WORLD,
               &status);
        }
        if (neighbor2 != NONE)
        {
                MPI_Send(&u[iz][offset+rows-1][0], NYPROB, MPI_FLOAT,
               neighbor2, NGHBOR1, MPI_COMM_WORLD);
                source = neighbor2;
                msgtype = NGHBOR2;
```

```
                    MPI_Recv(&u[iz][offset+rows][0], NYPROB, MPI_FLOAT,
                    source, msgtype, MPI_COMM_WORLD, &status);
                }
                /* Now call update to update the value of grid points */
                update(start, end, NYPROB, &u[iz][0][0], &u[1-iz][0][0]);
            /* Finally, send my portion of final result back to master */
        MPI_Send(&offset, 1, MPI_INT, MASTER, DONE,
        MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, MASTER, DONE,
        MPI_COMM_WORLD);
            MPI_Send(&u[iz][offset][0], rows*NYPROB, MPI_FLOAT, MASTER,
        DONE, MPI_COMM_WORLD);
        }
          //  MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
}

/**************************************************************************
*****
 * Initialize grid.
 *
*************************************************************************
***/
inidat(nx, ny, u);
prtdat(nx, ny, u, "inidat");
for (ix = 0; ix <= nx-1; ix++)
```

```
{
        u[1][ix][0] = u[0][ix][0];
        u[1][ix][ny-1] = u[0][ix][ny-1];
}
for (iy = 0; iy <= ny-1; iy++)
{
        u[1][0][iy] = u[0][0][iy];
        u[1][nx-1][iy] = u[0][nx-1][iy];
}

/**********************************************************************
 * Iterate over all timesteps.
 * ********************************************************************/
for (it = 1; it <= STEPS; it++)
{
        update(nx, ny, &u[iz][0][0], &u[1-iz][0][0]);
        iz = 1 - iz;
        prtdat(nx, ny, &u[iz][0][0], "final.dat");
}
/**********************************************************************
 * subroutine update
 *
**********************************************************************/
void update(int start, int end, float u[0][0][0])
{
        int ix, iy, iz;
```

```
        int nx;
        int ny ;
        for (ix = start; ix <= end; ix++)
        {
                for (iy = 1; iy <= ny-2; iy++)
                {
                        u[1-iz][ix][iy] = u[iz][ix][iy] +
                                parms.cx*(u[iz][ix-1][iy] + u[iz][ix+1][iy] -
                                        2*(u[iz][ix][iy])) +
                                parms.cy*(u[iz][ix][iy-1] + u[iz][ix][iy+1] -
                                        2*(u[iz][ix][iy]));
                }
        }
}

/*********************************************************************
**
 * subroutine inidat
 *
*********************************************************************
*/
void inidat(int nx, int ny, float u[nx][ny])
{
        int ix, iy;
        int nx;
        int ny;
```

```
        for (ix = 0; ix <= nx-1; ix++)
        {
                for (iy = 0; iy <= ny-1; iy++)
                {
                        u[ix][iy];
                }
        }
}

/**************************************************************************
***
 * subroutine prtdat
 *
**************************************************************************
**/
void prtdat(int nx, int ny, float u[nx][ny], char *fnam)
{
        int ix, iy;
        int nx;
        int ny;
        FILE *fp;
        fp = fopen(fnam, "w");
        for (iy = ny-1; iy >= 0; iy--)
        {
                for (ix = 0; ix <= nx-1; ix++)
                {
```

```
                    fprintf(fp, "%8.3f", u[ix][iy]);
                    if (ix != nx-1)
                        fprintf(fp, " ");
                    else
                        fprintf(fp, "\n");
                }
            }
            fclose(fp);
}
```

APPENDIX D

Derivative by Finite Difference Method for Various Dimensions with Theorems

The finite difference method is based on the local approximations of the partial derivatives in a PDE (Jain (1984)), which is derived by low order Taylor series expansions. The matrices that result from this discretizations are often well structured, which means that they typically consist of a non-zero diagonals. The general form of a PDE:

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} + d\frac{\partial u}{\partial x} + e\frac{\partial u}{\partial y} + g = 0 \qquad \text{(D.1)}$$

Eq.(D.1) is called Parabolic if $b^2 - 4ac = 0$.

Heat equation diffusion problems are examples of parabolic equations (Saulev (1964)). Methods to solve parabolic equations are: (a) Explicit Method and (b) Implicit Method.

If the boundary conditions are function values at the end point then the problem is known as Dirichlet Problem (DP)**.** If one or both boundary condition are derivatives values, then the problem is known as Neumann Problem (NP). Suppose that $U = U(x, y)$, the value of u at the two points (x, y) and $(x + h, y + k)$ are related by the Taylor's expansion:

$$U(x+h, y+k) = U(x,y) + (\frac{h\partial + k\partial}{\partial x \partial y})U(x,y) + \frac{1}{2!}\left(\frac{h\partial + k\partial}{\partial x \partial y}\right)^2 U(x,y) + \ldots + \frac{1}{(n-1)!}\left(\frac{h\partial + k\partial}{\partial x \partial y}\right)^{n-1} U(x,y) + R_n \tag{D.2}$$

where the remainder term is given by:

$$R_n = \frac{1}{n!}\left(\frac{h\partial + k\partial}{\partial x \partial y}\right)^n U(x+\xi h, y+\xi k), \qquad 0 < \xi < 1 \tag{D.3}$$

that is, $R_n = 0(|h| + |k|)^n$ (D.4)

by (2.4) there exist a positive constant $M \in |R_n| \le M(|h| + |k|)^n$ as both h and k tends to zero. The space point $(i\Delta x, j\Delta y)$ is called the grid points. Expanding in Taylor series for $U_{i-1,j}$ and $U_{i+1,j}$ about the central value $U_{i,j}$, we obtain:

$$U_{i-1,j} = U_{i,j} - (\Delta x)U_x + (\Delta x)^2 \frac{U_{xx}}{2!} - (\Delta x)^3 \frac{U_{xxx}}{3!} + (\Delta x)^4 \frac{U_{xxxx}}{4!}$$

$$U_{i+1,j} = U_{i,j} + (\Delta x)U_x + (\Delta x)^2 \frac{U_{xx}}{2!} + (\Delta x)^3 \frac{U_{xxx}}{3!} + (\Delta x)^4 \frac{U_{xxxx}}{4!}$$

where $U_x = \frac{\partial u}{\partial x}$, $U_{xx} = \frac{\partial^2 u}{\partial x^2}$, etc. and all derivatives are evaluated at the grid point (i, j). By taking these equations singly and by adding or subtracting one from another, we obtain the following finite-difference formula for the first and second derivatives at (i, j):

$$\frac{\partial u}{\partial x} = \frac{U_{i+1,j} - U_{i,j}}{(\Delta x)} + O(\Delta x) \tag{D.5}$$

$$\frac{\partial u}{\partial x} = \frac{U_{i,j} - U_{i-1,j}}{(\Delta x)} + O(\Delta x) \tag{D.6}$$

$$\frac{\partial u}{\partial x} = \frac{U_{i+1,j} - U_{i-1,j}}{2(\Delta x)} + O(\Delta x)^2 \tag{D.7}$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{(\Delta x)^2} + O(\Delta x)^2 \tag{D.8}$$

formula (D.5) - (D.7) are known as the forward, backward and central difference forms respectively. For convenience, the central-difference operator δx will be used occasionally. It is defined by:

$$U_{i,j} = \frac{U_{i+1/2,j} - U_{i-1/2,j}}{\Delta x^2} \tag{D.9}$$

whence $\delta x\, U_{i,j} = \dfrac{U_{i-1,j} - 2U_{i,j} + U_{i+1,j}}{(\Delta x)^2}$ (D.10)

by Saulev (1964),

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}, \quad for \quad 0 < x < 1, \quad 0 < t < T \tag{D.11}$$

$$\left.\begin{aligned} U(x,0) &= f(x), && 0 \le x \le 1 \\ U(0,t) &= g_o(t), && 0 < t \le T \\ U(1,t) &= g_1(t), && 0 < t \le T \end{aligned}\right\} \tag{D.12}$$

Explicit Method

Our approach to solving Parabolic Partial Differential Equations by a numerical method is to replace the Partial derivatives by finite-difference approximations. For the one dimensional heat flow equation (McDonough (1994))

$$\frac{\partial U}{\partial t} = \frac{k}{bc\rho}\frac{\partial^2 U}{\partial x^2} \tag{D.13}$$

we can use the relations

$$\left.\frac{\partial u}{\partial t}\right|_{t=t_j}^{x=x_i} = \frac{U_i^{j+1} - U_i^j}{\Delta t} + O\Delta t \tag{D.14}$$

and

$$\frac{\partial^2 u}{\partial x^2} = \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{(\Delta x)^2} + O(\Delta x)^2 \tag{D.15}$$

we use subscripts to denote position and superscript for time. Note that the error terms are of different orders, since a forward difference is used in equation (D.13). This introduces some special limitations, but it does simplify the procedure. Substituting

(d.14) and (D.15) into (D.13) and solving for $U_{i,j+1}$ gives the equation for the forward-difference method:

$$U_{i,j+1} = \frac{k\Delta t}{c\rho(\Delta x)^2}\left(U_{i+1,j} + U_{i-1,j}\right) + \left(1 - 2k\,\Delta t/c\rho(\Delta x)^2\right)U_{i,j} \qquad \text{(D.16)}$$

we have solved for $U_{i,j+1}$ in terms of the temperatures at time t_j in equation (D.16) in view of the normally known conditions for a Parabolic PDE. We subdivide the length into uniform subintervals, and apply the finite difference approximation to equation (D.13) at each point where u is not known, equation (D.16) then gives the values of u at each point at $t = t_1$, since the values at $t = t_0$ are given by the initial conditions. It can be used to get values at t_2 using the values at t_1 as initial conditions, so we can step the solution forward in time. At the endpoints, the boundary conditions will determine U. If the ratio of $k\Delta t/(\Delta x)^2 = r$ is chosen so that $k\Delta t/c\rho(\Delta x)^2 = 1/2$, the equation is simplified in that the last term vanishes and we have:

$$U_{i,j+1} = \frac{1}{2}(U_{i+1,j} + U_{i-1,j}) \qquad \text{(D.17)}$$

The value of r is critical, if the value of r is chosen as less than one-half there will be improved accuracy and this is called a marching method. Similarly, if the value of r is greater than one-half which would reduce the number of calculations required to advance the solution through a given interval of time, the phenomenon of instability sets in. The forward difference method is an explicit method since all the approximations can be found directly based on the information from the initial and boundary conditions. These conditions give us values at $(x_o, t_o), (x_1, t_1), (x_2, t_2)$, from which we get an approximation at (x_1, t_1). Adding the initial condition at (x_3, t_o) gives us an approximation at (x_2, t_1), and so on across the row.

Implicit Method of Solution (Carnahan et. al., 1969)

The explicit schemes outlined are simple to implement however they suffer from one major problem. They typically only provide useful solutions when

$$\Delta t \leq \frac{(\Delta x)^2}{2} \tag{D.18}$$

Thus, as the spatial grid Δx is refined to improve accuracy, the number of calculations to reach any fixed time t_f increases enormously. For instance, for a particular problem a grid 10 times as fine would required 100 times more steps and therefore 1000 times more calculations. This deficiency can be a serious problem when solving multi-dimensional equations. If $U_{i,n}$ is known at a grid point then $U_{i,n}$ can be calculated at every point. In performing this calculation, knowledge of the value of $U_{i,n}$ on ER and FQ is not required. For fixed $r = \Delta t/(\Delta x)^2$ as $\Delta x \to 0$, the slope of EP $\Delta t = r\Delta x \to 0$ and hence in the limit EP tends to the true characteristics. Consequently, for finite Δx the explicit finite difference approximation is an imperfect model for a parabolic equation, although for $\Delta t << \Delta x$ the gradient $\frac{\Delta t}{\Delta x} \to 0$. It is possible to bypass the limitation (D.18)) and produce a scheme whereby boundary conditions are required to advance the solution to the next point. We approximate the time derivatives in (D.11) by the backward difference

$$\frac{\partial u}{\partial t} \approx U_{i,j} - U_{i,j-1} \tag{D.19}$$

The truncation error in this case is $O(\Delta t)$. Substituting Eq.(D.18) and Eq.(D.19) into Eq.(D.11) gives us the following finite difference representation:

$$\frac{U_{i,n} - U_{i,n-1}}{\Delta t} = \frac{U_{i+1,n} - 2U_{i,n} + U_{i-1,n}}{(\Delta x)^2} \tag{D.20}$$

In this case each value of U at time level n depends upon other values at this time level as well as values at time level $n-1$.

Rearranging (D.20) and writing in terms of time levels $n+1$ and n. In this way it can be said that the PDE $U_t = U_{xx}$ is approximated at the midpoint M with

$$U_{xx} = \frac{1}{2(\Delta x)^2}(U_{i+1,n} - 2U_{i,n} + U_{i-1,n+1} - 2U_{i,n+1} + U_{i-1,n+1}) \tag{D.21}$$

$$U_t = \frac{1}{\Delta t}(U_{i,n+1} - U_{i,n}) \tag{D.22}$$

by equating the two approximations and multiplying by 2k and subsequent ordering, we obtain the following difference equation for an interior point. Rearranging Eq.(D.21) and Eq.(D.22) in Eq.(D.11) we have:

$$-rU_{i-1,n+1} + (2+2r)U_{i,n+1} - rU_{i+1,n+1} = rU_{i-1,n} + (2-2r)U_{i,n} + rU_{i+1,n} \tag{D.23}$$

the following conditions are based upon the initial boundary value problem Eq.(D.11) and Eq.(D.12). The above is called an implicit finite-difference scheme as we can not explicitly write $U_{i,n+1}$ $(i = 0,\cdots,m)$ in terms of known quantities. We can write Eq.(D.23) in matrix form as:

$$BU_{n+1} = U_n + d_n \tag{D.24}$$

where,

$$U_n = \left[U_1, U_2, \cdots, U_{m-1,n}\right]^T,$$
$$d_n = \left[rU_{o,n+1}, 0, \cdots, 0\ rU_{m,n+1}\right]^T,$$

and B is the $(m-1)\times(m-1)$ matrix

$$B = \begin{bmatrix} (1+2r) & -r & & & \\ -r & (1+2r) & -r & & \\ & & \cdots & & \\ & & -r & (1+2r) & -r \\ & & & -r & (1+2r) \end{bmatrix}$$

multiplying both side of Eq. (D.24) by B^{-1} gives

$$U_{n+1} = B^{-1}U_n + B^{-1}d_n \tag{D.25}$$

thus, if we can invert the matrix B we can advance the solution forward in time. In general matrix inversion is a difficult problem. However in the numerical solution of PDE we often come across sparse matrices or matrices with some special structure. The matrix B is a tridiagonal matrix which we can invert using the Thomas Algorithm (Carnahan et. al., (1969)).

Theorem D.1 (McDonough (1994))

A necessary and sufficient condition for convergence of the iteration (D.29) - (D.32) from any initial guess is:

$$\rho(G) < 1, \tag{D.38}$$

where

$$\rho(G) = Max|\lambda_i|, \ \lambda \in \sigma(G), \ 1 \le i \le N \tag{D.39}$$

is the spectra radius of the iteration matrix G, and $\sigma(G)$ is the notation for the spectrum (set of all eigenvalues) of G. We remark (without proof) that this basically follows from the contraction mapping principle and the fact that

$$\rho(G) \le \|G\| \tag{D.40}$$

for all norms, $\|.\|$. We also note that convergence may occur even when $\rho(G) \le 1$ holds, but only restricted set of initial guesses. It should be clear that $\rho(G)$ corresponds to the Lipchitz constant.

Definition D.1: The residual after n iterations is:

$$r_n = b - Au^{(n)} \tag{D.41}$$

Definition D.2: The exact error after n iterations is:

$$e_n = u - u^{(n)} \tag{D.42}$$

Definition D.3: The iteration error after n iterations is:

$$d_n = u^{(n+1)} + u^{(n)} \tag{D.43}$$

Hence, r_n and e_n are related by:

$$Ae_n = r_n \tag{D.44}$$

from Eq.(D.33) and definition (D.32) that

$$e_n = Ge_{n-1} = G^2 e_{n-2} = \cdots = G_n e_0 \tag{D.45}$$

and similarly for d_n.

Definition D.4: The average convergence rate for iterations of (3.32) is given by

$$R_n(G) = \frac{-1}{n}\log\|G_n\| \tag{D.46}$$

Definition D.5: The asymptotic convergence rate is defined as

$$R_\infty(G) \equiv -\log \rho(G) \tag{D.47}$$

It is the asymptotic convergence that is more important gauging performance of iterative methods when they are to be used to produce highly-accurate solutions. Values of (D.45) depend only on the spectral radius of the iteration matrix and are thus unique. It is shown that

$$R_\infty(G) = \lim_{n\to\infty} R_n(G) \tag{D.48}$$

This relationship is used in obtaining estimates of total arithmetic required by iterative methods. Hence, the iterative error and exact error are related by:

$$d_n = (1-G)e_n \tag{D.49}$$

or

$$e_n = (1-G)^{-1} d_n \tag{D.50}$$

then

$$\|e_n\| \le \|(1-G)^{-1}\|\|d_n\| \tag{D.51}$$

$$\|e_n\| \le \frac{1}{1-\rho(G)}\|d_n\|, \quad \rho(G) < 1 \tag{D.52}$$

Theorem D.2 (Lee & Riess (1991)): Let e_n and d_n be as defined in Eq.(D.42) and Eq.(D.43) respectively, and suppose $\rho(G) > 1$ holds. Then for any norm $\| \ \|$,

$$\lim_{n\to\infty} \frac{\|d_n\|}{\|d_{n-1}\|} = \rho(G), \text{and } \lim_{n\to\infty} \frac{\|e_n\|}{\|d_{dn-1}\|} = \frac{1}{\|1-\rho(G)\|}.$$

recall that $d_n = Gd_{n-1}$, which implies $\|d_n\| \le \|G\|\|d_{n-1}\|$ for compatible norms. In particular, if we take the vector norm to be 2-norm we may use the spectra norm as the matrix norm. Then if G is diagonalizable we have

$$\|G\| = \rho(G)$$

thus,

$$\frac{\|d_n\|}{\|d_{n-1}\|} = \rho(G)$$

employing the Reyleigh quotient to arrive at nth reverse inequality

$$\rho(G) =< \frac{d_{n-1,} Gd_{n-1}}{\|d_{n-1}\|^2} >=< \frac{d_{n-1,} d_n}{\|d_{n-1}\|^2}$$

but Cauchy-Schwarz inequality follows that:

$$\rho(G) \le \frac{\|d_n\|\|d_{n-1}\|}{\|d_{n-1}\|^2} = \frac{\|d_n\|}{\|d_{n-1}\|},$$

recall from Eq.(D.52) that:

$$\frac{\|e_n\|}{\|d_n\|} \le \frac{1}{1-\rho(G)}$$

from Eq.(D.49), we have:

$$(1-G)e_n = d_n \text{ and } \|(1-G)e_n\| = \|d_n\| \text{ which implies } \|d_n\| \le \|(1-G)\|\|e_n\|,$$

again, for compatible matrix and vector norms, using the matrix spectral norm and the vector 2-norm, respectively, gives:

$$\frac{\|e_n\|}{\|d_n\|} \ge \frac{1}{1-\rho(G)}$$

Background of the Fourth Order in Space

According to Jennifer et al., (2007), the problem which we considered here is the numerical solution of Eq. (D.11). Let Δx and Δt denote spatial mesh size and time increment, respectively. We assume that there exists an integer M, such that $(M+1)\Delta x = 1$, and $U_{i,j}$ and $(Uxx)_{i,j}$ are used to represent the numerical approximations of $U(i\Delta x, j\Delta t)$ and $Uxx(i\Delta x, j\Delta t)$, respectively. Also, we use i, j to represent the space and time indexes and $\Delta t = T/(n+1)$. The mesh ratio is taken as $\lambda = \Delta t/(\Delta x)^2$ where $0 \le i \le j+1$ and $j \ge 0$.

The application of the well-known Crank-Nicolson type of scheme to Eq. (D.11) results in the following expression at the point $(x_i, t_{j+1/2})$ is

$$-\lambda\theta U_{i-1,j+1} + (1+2\lambda\theta)U_{i,j+1} - \lambda\theta U_{i+1,j+1} = \lambda(1-\theta)U_{i-1,j} + [1-2\lambda(1-\theta)]U_{i,j} + \lambda(1-\theta)U_{i+1,j} \qquad i = 1,2,\ldots M \tag{D.53}$$

the above approximation corresponds to the fully implicit, the Crank-Nicolson and the classical explicit methods when θ takes the values 1, ½, and 0 respectively and Eq. (D.53) is of order $O(\Delta t)^2$ in time . Sahimi et al., (2001) developed the Iterative Alternating Decomposition Explicit (IADE) method to solve (D.11). The second-order IADE scheme entailed the decomposition of a tridiagonal matrix which arises from the difference method used to approximate the parabolic equation. By employing the fractional scheme of Yanenko (Yanenko (1971)) and the Mitchell-Fairweather (MF) variant, this method proves to be highly accurate, fast, convergent and stable. Here, we will construct fourth-order approximations to the terms $(Uxx)_{i,j}$ and $(Uxx)_{i,j+1}$, so that Eq. (D.53) is order $O(\Delta x^4)$ in space.

The Interior Points (Jennifer et al., (2007))

The first part of our construction is for all interior points (x_i, t_j), where $2 \le i \le M-1$ and $M \ge 1$. The following compact scheme is used in order to derive a fourth-order approximation to $(Uxx)_{i,j}$:

$$\frac{U(x_{i+1}, y_j) - 2U(x_i, y_j) + U(x_{i-1}, y_j)}{\Delta x^2} = aU_{xx}(x_{i+1}, y_j) + bU_{xx}(x_i, y_j) + cU_{xx}(x_{i-1}, y_j) \tag{D.54}$$

where a, b and c are constants to be determined. The Taylor series expansions to terms $U(x_{i+1}, y_j)$ and $U(x_{i-1}, y_j)$ on the left side of Eq. (D.54), at the point (x_i, y_j) yield the following result:

$$\frac{U(x_{i+1}, y_j) - 2U(x_i, y_j) + U(x_{i-1}, y_j)}{\Delta x^2} = U_{xx}(x_i, y_j) + \frac{\Delta x^2}{12} U_{xxxx}(x_i, y_j) + O(\Delta x^4). \tag{D.55}$$

Similarly, expansions for $U_{xx}(x_{i+1}, y_j)$ and $U_{xx}(x_{i-1}, y_j)$, on the right side of Eq. (D.55) at the same point yield the result below:

$$U_{xx}(x_{i+1}, y_j) = U_{xx}(x_i, y_j) + \Delta x U_{xxx}(x_i, y_j) + \frac{\Delta x^2}{2} U_{xxxx}(x_i, y_j) + \frac{\Delta x^3}{3!} U_{xxxxx}(x_i, y_j) + O(\Delta x^4) \tag{D.56}$$

the substitution of Eq. (D.54) into Eq. (D.55) and Eq. (D.56) give the next results:

$$U_{xx}(x_i, y_j) + \frac{\Delta x^4}{12} U_{xxxx}(x_i, y_j) + O(\Delta x^4) = (a+b+c)U_{xx}(x_i, y_j) + (-a+c)\Delta x U_{xxx}(x_i, y_j) + \frac{a+c}{2}\Delta x^2 U_{xxxx}(x_i, y_j) + \frac{-a+c}{6}\Delta x^3 U_{xxxx}(x_i, y_j) + O(\Delta x^4). \tag{D.57}$$

To turn both sides of Eq. (D.57) into fourth-order, we need to equate the corresponding coefficients for terms involving $\Delta x^o, \Delta x, \Delta x^2$ and Δx^3 on both sides. This result in the following set of linear equations in terms of a, b and c :

$$a+b+c=1,\ \ a-c=0,\ \frac{a+c}{2}=\frac{1}{12},\ \frac{a-c}{6}=0,$$

for which the solutions are

$$a=c=\frac{1}{12},\ b=\frac{5}{6}$$

substituting these values into and Eq. (D.54) and changing

$U(x_{i-1},y_j),U(x_i,y_j),U(x_{i+1},y_j)$ and $U_{xx}(x_{i-1},y_j),U_{xx}(x_i,y_j)$ and $U_{xx}(x_{i+1},y_j)$ to their analogous notations, the following fourth-order relation at all interior points and any time level is derived.

$$\frac{1}{10}(U_{xx})_{i-1,j}+(U_{xx})_{i,j}+\frac{1}{10}(U_{xx})_{i+1,j}=\frac{6}{5\Delta x^2}(U_{i-1,j}-2U_{i,j}+U_{i+1,j}),\quad 2\le i\le M-1,\ M\ge 1. \tag{D.58}$$

It can be seen that Eq. (D.58) is obtained from Taylor series only:

$$(U_{xx})_i=\frac{U_{i+1}-2U_i+U_{i-1}}{\Delta x^2}+O(\Delta x^2) \tag{D.59}$$

The Boundary Points (Jennifer et al., (2007))

We now develop similar fourth-order approximations for $(U_{xx})(x_1,y_j)$ and $U_{xx}(x_M,y_j)$, where x_1 and x_M are the two points next to the actual boundary points 0 and 1. We start with (x_1,y_j) first, where the following combined scheme is used:

$$\frac{b^*}{\Delta x}U_x(0,y_j)+a^*U_{xx}(x_1,y_j)+c^*U_{xx}(x_{2,}y_j)=\frac{1}{\Delta x^2}(e^*U(x_1,y_j)+f^*U(x_2,y_j), \tag{D.60}$$

a^*,b^*,c^*,e^* and f^* are constants to be determined. By multiplying both sides of Eq. (D.60) by Δx^2 we can rewrite it as follows:

$$b^*\Delta xU_x(0,y_j)+a^*\Delta x^2U_{xx}(x_1,y_j)+c^*\Delta x^2U_{xx}(x^2,y_j)=e^*U(x_1,y_j)+f^*U(x_2,y_j) \tag{D.61}$$

if each term of Eq. (D.61) is expanded in Taylor series at point $(0, y_j)$, we would obtain the following results:

$$U(x_1, y_j) = U(0, y_j) + \Delta x U_x(0, y_j) + \frac{\Delta x^2}{2} U_{xx}(0, y_j) + \frac{\Delta x^3}{6} U_{xxx}(0, y_j) + O(\Delta x^4) \quad \text{(D.62)}$$

$$U(x_2, y_j) = U(0, y_j) + 2\Delta x U_x(0, y_j) + \frac{(2\Delta x)^2}{2!} U_{xx}(0, y_j) + \frac{(2\Delta x)^3}{3!} U_{xxx}(0, y_j) + O(\Delta x^4) \quad \text{(D.63)}$$

$$U_{xx}(x_1, y_j) = U_{xx}(0, y_j) + \Delta x U_{xxx}(0, y_j) + O(\Delta x^2) \quad \text{(D.64)}$$

$$U_{xx}(x_2, y_j) = U_{xx}(0, y_j) + 2\Delta x U_{xxx}(0, y_j) + O(\Delta x^2). \quad \text{(D.65)}$$

The substitution into Eq. (D.61) of Eq. (D.62-D.65) and afterward simplifications give the following result:

$$\begin{aligned} &b^* \Delta x U_x(0, y_j) + (a^* + c^*)\Delta x^2 U_{xx}(0, y_j) + (a^* + 2c^*) U_{xxx}(0, y_j) + O(\Delta x^4) = \\ &(e^* + f^*) U(0, y_j) + (e^* + 2f^*)\Delta x U_x(0, y_j) + (e^* + 2f^*)\Delta x U_{xx}(0, y_j) + \\ &\frac{e^* + 8f^*}{6} \Delta x^3 U_{xxx}(0, y_j) + O(\Delta x^4) \end{aligned} \quad \text{(D.66)}$$

to derive a fourth-order approximation in Eq. (D.66), we only need to equate the corresponding coefficients for those terms involving $\Delta x^o, \Delta x, \Delta x^2$ and Δx^3 on both sides. This results in the following system equations:

$$e^* + f^* = 0,\ b^* = e^* + 2f^*,\ a^* + c^* = \frac{e^*}{2} + 2f^*,\ a^* + 2c^* = \frac{e^* + 8f^*}{6}, \quad \text{(D.67)}$$

with the solutions to be

$$a^* = \frac{11}{6} f^*,\ b^* = f^*,\ c^* = -\frac{1}{3} f^*,\ e^* = -f^* \quad \text{(D.68)}$$

we take $f^* = 1$ in the above solutions, and substitute it together with the corresponding values of a^*, b^*, c^*, d^*, e^* into Eq. (D.61) get the following scheme:

$$\frac{11}{6} U_{xx}(x_1, y_j) - \frac{1}{3} U_{xx}(x_2, y_j) = \frac{-1}{\Delta x} U_x(0, y_j) + \frac{U(x_2, y_j) - U(x_1, y_j)}{\Delta x} \quad \text{(D.69)}$$

after replacing $U(x_i, y_j)$ to $U_{i,j}$ and applying the boundary condition $U_x(0, yj) = \alpha_1(y_j)$, the following fourth-order scheme at the boundary point (x_1, y_j) is obtained:

$$\frac{11}{6}(U_{xx})_{i,j} - \frac{1}{3}(U_{xx})_{2,j} = \frac{-\alpha_1(y_j)}{\Delta x} + \frac{U_{2,j} - U_{1,j}}{\Delta x^2}. \qquad \text{(D.70)}$$

Similarly, the following fourth-order scheme at the point (x_{M-1}, y_j) can also be derived:

$$\frac{11}{6}(U_{xx})_{M,j} - \frac{1}{3}(U_{xx})_{M-1,j} = \frac{\alpha_2(y_j)}{\Delta x} + \frac{U_{M-1,j} - U_{M,j}}{\Delta x^2}. \qquad \text{(D.71)}$$

Thus, the set of schemes, consisting of Eq. (D.53), (D.58), (D.70) and (D.71) have overall order of $O(\Delta t^2 + \Delta x^4)$, which indicates that our set of schemes are consistent with the differential equation.

Unconditional Stability for 2-D Telegraph Equation

The general way to verify the stability of a finite-difference kind algorithm is to put an elemental solution into the algorithm and make sure that the amplitude of the propagation gain is no more than one. By applying the Von Neumann analysis as in Smith, (1985), we can analytically prove that the 2-D ADI method is unconditionally stable. Consider the elemental solution of Eq. (D.4)

$$v_{i,j}^n = K^n e^{I(ik_x\Delta x + jk_y\Delta y + kk)} \qquad \text{(D.72)}$$

where k_x *and* k_y are the wave numbers along the x *and* y direction, respectively, and k is propagation gain. Putting this elemental solution into the 2-D ADI algorithm, and with some manipulations, we get:

$$K^2 - \frac{2(R_x R_y + c_o)}{(1+R_x)(1+R_y)}K + \frac{R_x R_y + c_1}{(1+R_x)(1+R_y)} = 0 \qquad \text{(D.73)}$$

where

$$R_x = 4\rho_x \sin^2(k_x \Delta x/2) \tag{D.74}$$

$$R_y = 4\rho_y \sin^2(k_y \Delta y/2) \tag{D.75}$$

The solutions of Eq. (D.73) are equal to:

$$K = \frac{A + c_o \pm \sqrt{D}}{(1+R_x)(1+R_y)} \tag{D.76}$$

where

$$D = \sqrt{(A+c_0)^2 - (1+R_x)(1+R_y)(A+c_1)}$$

$$A = R_x R_y.$$

By examining the amplitude of K, we are able to prove that the 2-D ADI algorithm is unconditional stable in the following theorem.

Theorem D.3*:* The 2-D ADI algorithm is unconditionally stable.

Proof: To prove that the 2-D ADI method is unconditionally stable, we need show the amplitude of the gain factor K is less than or equal to K one. Let us consider the following two cases.

Case 1: $D \geq 0$.

From Eq. (5.28 – 5.31), we know that $A + c_0$ is greater or equal to zero. Hence

$$|K| \leq \frac{A + c_0 + \sqrt{D}}{(1+Rx)(1+Ry)}$$
$$= \frac{A + c_0 + \sqrt{D}}{A + c_0 + R_x + R_y + c_0 - c_1}.$$

We only need to prove $D \leq (R_x + R_y + c_0 - c_1)^2$, since $c_0 - c_1$ is also greater than zero

$$D - (R_x + R_y + c_0 - c_1)^2 = -(R_x + R_y)(1+R_x)(1+R_y) \leq 0.$$

Therefore, $|K| \leq 1$.

Case 2: $D \le 0$.

$$
\begin{aligned}
|K|^2 &= \frac{A + c_1}{(1+R_x)(1+R_y)} \\
&= \frac{A + \dfrac{\dfrac{1}{(\Delta t)^2} - \dfrac{a}{2\Delta t}}{\dfrac{1}{(\Delta t)^2} + \dfrac{a}{2\Delta t}}}{A + 1 + R_x + R_y} \\
&\le 1.
\end{aligned}
$$

Therefore, the 2-D ADI method is unconditionally stable from the above derivations.

Linear Runtime

There are two sub iterations need to be preformed for each time step. By analysis the runtime of each sub iteration, we are able to prove the computational load of the 2-D ADI algorithm is linear time at each time step in the following theorem.

Theorem D.4: The runtime of the 2-D ADI algorithm is *O(N)* at each time step, where $N = N_x x N_y$ is the number of total nodes.

Proof: Let us consider Sub iteration 1. We can divide the set of these *N* nodes by $N_x x N_y$ subsets with each one containing N_x points in the *x* direction. Since only two unknown variables need to be solved in the updating equation with each (i, j), the coefficient matrix $\phi_{i,j}$ associated with updating $v'_{.,j}$ s is a triangular matrix at each subset. Therefore, the runtime of the updating $v'_{.,j}$ s is linear with $O(N_x)$. There are $N_x x N_y$ subsets in sub iteration 1. Hence, the computational load of the sub iteration 1 is

$O(N_x x N_y)$ at each time step. The runtime of sub iteration 2 is also $O(N)$ in a similar way. Hence, the total runtime of the 2-D ADI algorithm is $O(N)$ at each time step.

Stability Analysis for 3-D ADI Telegraphic Equation (Mohanty, (2009))

With reference to (Mohanty et. al. (2004)) we can analytically prove that the 3-D ADI method is unconditionally stable. Consider the elemental solution of Eq. (D.103)

$$v_{i,j,k}^{n} = K^{n} e^{I(ik_x\Delta x + jk_y\Delta y + kk_z\Delta z)} \tag{D.77}$$

where k_x, k_y and k_z are the wave numbers along the x, y and z direction, respectively, and k is propagation gain. Putting this elemental solution into the 3-D ADI algorithm, and with some manipulations, we get:

$$K^2 - \frac{2(R_xR_y + R_yR_z + R_zR_x + R_xR_yR_z + c_o)}{(1+R_x)(1+R_y)(1+R_z)} K + \frac{R_xR_y + R_yR_z + R_zR_x + R_xR_yR_z + c_1}{(1+R_x)(1+R_y)(1+R_z)} = 0 \tag{D.78}$$

where

$$R_x = 4\rho_x \sin^2(k_x\Delta x/2) \tag{D.79}$$

$$R_y = 4\rho_y \sin^2(k_y\Delta y/2) \tag{D.80}$$

$$R_z = 4\rho_z \sin^2(k_z\Delta z/2) \tag{D.81}$$

The solutions of Eq. (D.78) are equal to:

$$K = \frac{A + c_o \pm \sqrt{D}}{(1+R_x)(1+R_y)(1+R_z)} \tag{D.82}$$

where

$$D = \sqrt{(A+c_0)^2 - (1+R_x)(1+R_y)(1+R_z)(A+c_1)}$$

$$A = R_xR_y + R_yR_z + R_zR_x + R_xR_yR_z.$$

by examining the amplitude of K, we are able to prove that the 3-D ADI algorithm is unconditional stable in the following theorem.

Theorem D.5 The 3-D ADI algorithm is unconditionally stable (Mohanty, (2009)).

With reference to (Mohanty et al., (2004) and Mohanty (2009)), to prove that the 3-D ADI method is unconditionally stable, we need show the amplitude of the gain factor K is less than or equal to one. Let us consider the following two cases.

Case 1: $D \geq 0$.

From Eq. (D.78) and Eq. (D.79) – Eq. (D.81), we know that $A + c_0$ is greater or equal to zero. Hence

$$|K| \leq \frac{A + c_0 + \sqrt{D}}{(1+Rx)(1+Ry)(1+Rz)}$$
$$= \frac{A + c_0 + \sqrt{D}}{A + c_0 + R_x + R_y + R_z + c_0 - c_1}.$$

we only need to prove $D \leq (R_x + R_y + R_z + c_0 - c_1)^2$, since $c_0 - c_1$ is also greater than zero

$$D - (R_x + R_y + R_z + c_0 - c_1)^2 = -(R_x + R_y + R_z)(1 + R_x)(1 + R_y)(1 + R_z) \leq 0.$$

Therefore, $|K| \leq 1$.

Case 2: $D \leq 0$.

$$|K|^2 = \frac{A + c_1}{(1+R_x)(1+R_y)(1+R_z)}$$
$$= \frac{A + \dfrac{1}{(\Delta t)^2} - \dfrac{a}{2\Delta t}}{A + 1 + R_x + R_y + R_z}$$
$$\leq 1.$$

Therefore, the 3-D ADI method is unconditionally stable from the above derivations.

Linear Runtime

Mohanty (2009) presented three sub iterations need to be preformed for each time step. By analysis the runtime of each sub-iteration as shown in Table 5.2, we are able to prove the computational load of the 3-D ADI algorithm is linear time at each time step in the following theorem.

Theorem D.6: The runtime of the 3-D ADI algorithm is $O(N)$ at each time step, where $N = N_x \times N_y \times N_z$ is the number of total nodes. Let us consider sub-iteration 1. We can divide the set of these *N* nodes by $N_y \times N_z$ subsets with each one containing N_x points in the *x* direction. Since only three unknown variables need to be solved in the updating equation with each (i, j, k), the coefficient matrix $\phi_{j,k}$ associated with updating $v'_{.,j,k}$ s is a triangular matrix as at each subset. Therefore, the runtime of the updating $v'_{.,j,k}$ s is linear with $O(N_x)$. There are $N_y \times N_z$ subsets in sub-iteration 1. Hence, the computational load of the sub iteration 1 is $O(N_x \times N_y \times N_z)$ at each time step.

The runtime of sub iteration 2 and 3 is also $O(N)$ in a similar way. Hence, the total runtime of the 3-D ADI algorithm is $O(N)$ at each time step.

$$\phi_{j,k} = \begin{bmatrix} \times & \times & 0 & \cdots & 0 \\ \times & \times & \times & \ddots & \vdots \\ 0 & \times & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \times & \times \\ 0 & \cdots & 0 & \times & \times \end{bmatrix} \tag{D.83}$$

APPENDIX E

Publications in International (ISI) Journals and Conferences

Ewedafe S. U. & Rio H. S. 2010. Armadillo Generation Distributed System and Geranium Cad cam Cluster for Solving 2-D Telegraphic Equation with MPI and PVM. *International Journal of Computer Mathematics*. Published http://prod.informaworld.com/smpp/title~db=all~content=g772621578. 2008 impact factor: 0.546 and 2008 cited half-life: 6.7 years (2008 Journal Citation Reports)

Ewedafe S. U. & Rio H. S. 2010. Parallel Implementation of 2-D Telegraphic Equation on MPI/PVM Cluster (Domain Decomposition Implementation). *International Journal of Parallel Programming*. Published http://www.springerlink.com/content/t5182031p6192817/. Impact factor: 0.875 of Journal Citation Reports®, Thomson Reuters.

Ewedafe S. U., & Rio H. S. 2010. Unconditional Stable 3-D Alternating Direction Implicit Method on 3-D *Telegraph Equation. International Journal of Mathematics and Engineering with Computers*. Accepted in December.

Ewedafe S. U & Rio H. S. 2008. Solving 2-D Parabolic Problems on two Distributed Platforms using Different Alternating Iterative Methods. *International Journal of Mathematics and Computer Science*. **3** (4): 247 – 278

Ewedafe S. U. & Rio H. S. 2009. Using Geranium Cad cam Cluster for Solving 2-D Bio-Heat Transfer Problem of the Iterative Alternating Direction Explicit Method with MPI. *5th Asian Mathematical Conference*. Putra World Trade Centre, Kuala Lumpur, Malaysia

Ewedafe S. U. & Rio H. S. 2009. Parallel Simulation of the Fourth-Order IADE Method on Heat-equation with PVM. *Proceedings of the 5th International Conference on Mathematics, Statistics and their Applications*. West Sumatera, Indonesia, June 9th – 11th

Ewedafe S. U. & Rio H. S. 2009. Solving 2-D Telegraphic Problem on Distributed System Using PVM. *3rd International Conference on Experiments/Process/System Modeling/Simulation/Optimization (3rd IC-EpsMsO)*. Greece, Athens 8th – 11th July