NAME – SOHAM CHATTOPADHYAY
SR NO. – 12404
MT-218 ASSIGNMENT 1.
DATE – 12th SEPTEMBER, 2016.

1. PROGRAM FOR SOLVING 2D DIFFUSION EQUATION (Diffusion Coefficient D = 1) WITH ADI METHOD USING JACOBI ALGORITHM FOR IMPLICIT DISCRETIZATION.

```
#include <stdio.h>
#include <stdlib.h>

#define MESHX 100
#define MESHY 100

#define ntime 2000
#define deltat 0.1
#define deltax 1.0
#define deltay 1.0
#define saveT 200
#define radius 20
double c_old[MESHX+2][MESHY+2];
double c_k[MESHX+2][MESHY+2];

void initialize();
void update();
void writetofile (FILE *fp);

void main ()
{
	long t;
	long x,y;
	char filename [1000];

	double a_x[MESHX+1];
	double b_x[MESHX+1];
	double c_x[MESHX+1];
	double d_x[MESHX+1];

	double a_y[MESHY+1];
	double b_y[MESHY+1];
	double c_y[MESHY+1];
	double d_y[MESHY+1];
	double inv_deltax2 = 1/(deltax*deltax);
	long iter;
```

```
	double alpha = inv_deltax2*deltat;

	FILE *fp;
	c_old[MESHX+2][MESHY+2];

	initialize();
	update();


	for (t=0;t<ntime;t++)
	{
		//implicit in x using jacobi method
		for (y=1;y<MESHY+1;y++)
		{
			d_x[1] = c_old[1][y] + alpha*c_old[0][y] + alpha*(c_old[1][y-1] -
2.0*c_old[1][y] + c_old[1][y+1]);
			a_x[1] = 0.0;
			b_x[1] = 1.0+2.0*alpha;
			c_x[1] = -alpha;
			for (x=2;x<MESHX+1;x++)
			{
				a_x[x] = -alpha;
				b_x[x] = 1.0+2.0*alpha;
				c_x[x] = -alpha;
				d_x[x] = c_old[x][y] + alpha*(c_old[x][y-1] - 2.0*c_old[x][y] +
c_old[x][y+1]);
			}
			c_x[MESHX]  = 0.0;
			d_x[MESHX] = c_old[MESHX][y] + alpha*c_old[MESHX+1][y] +
alpha*(c_old[MESHX][y-1] - 2.0*c_old[MESHX][y] + c_old[MESHX][y+1]);
			iter=0;
			update(c_old,c_k);
			for(iter=0;iter<10;iter++)
			 {
				for(x=1;x<MESHX+1; x++)
				{
					c_old[x][y] = (d_x[x] - a_x[x]*c_k[x-1][y] -
c_x[x]*c_k[x+1][y])/b_x[x];
				}
				update(c_old,c_k);
			 }
		}



		//implicit in y using jacobi method
		for (x=1;x<MESHX+1;x++)
			{
			d_y[1] = c_old[x][1] + alpha*c_old[x][0] + alpha*(c_old[x-1][1] -
2.0*c_old[x][1] + c_old[x+1][1]);
			a_y[1] = 0.0;
			b_y[1] = 1.0+2.0*alpha;
```

```
                        c_y[1] = -alpha;
                        for (y=2;y<MESHY+1;y++)
                        {
                                a_y[y] = -alpha;
                                b_y[y] = 1.0+2.0*alpha;
                                c_y[y] = -alpha;
                                d_y[y] = c_old[x][y] + alpha*(c_old[x-1][y] - 2.0*c_old[x][y] +
c_old[x+1][y]);
                        }
                        c_y[MESHY]  = 0.0;
                        d_y[MESHY] = c_old[x][MESHY] + alpha*c_old[x][MESHY+1] +
alpha*(c_old[x-1][MESHY] - 2.0*c_old[x][MESHY] + c_old[x+1][MESHY]);
                        iter=0;
                        update(c_old,c_k);
                        for(iter=0;iter<10;iter++)
                         {
                                for(y=1;y<MESHX+1; y++)
                                {
                                        c_old[x][y] = (d_y[y] - a_y[y]*c_k[x][y-1] -
c_y[y]*c_k[x][y+1])/b_y[y];
                                }
                                update(c_old,c_k);
                         }
            }
            if (t%saveT == 0)
            {
                        sprintf(filename,"composition%ld.dat", t);
                        fp=fopen (filename,"w");
                        writetofile (fp);
                        fclose (fp);
                }
        }
}


void initialize()
{
        long x,y;
        for (x=0;x<=(MESHX+1);x++)
        {
                for(y=0;y<=(MESHY+1);y++)
                {
                         if (((x-(MESHX+1)/2)*(x-(MESHX+1)/2) + (y-(MESHY+1)/2)*(y-
(MESHY+1)/2)) < radius*radius)
                                c_old[x][y] = 0.8;
                         else
                                c_old[x][y] = 0.1;
                }
        }
}
void update ()
{
```

```
	long x,y;
	for (y=0;y<MESHY+1;y++)
	{
		for (x=0;x<MESHX+1;x++)
		{
			c_k[x][y] = c_old[x][y];
		}
	}
}
void writetofile (FILE *fp)
{
 long x,y;
 for (x=0;x<=(MESHX+1);x++)
 {
	 for (y=0;y<=(MESHY+1);y++)
	 {
		 fprintf(fp, "%le %le %le\n", x*deltax, y*deltay, c_old[x][y]);
	 }
	 fprintf(fp,"\n");
 }
}
```

2. PROGRAM FOR SOLVING 2D DIFFUSION EQUATION (Diffusion Coefficient D = 1) WITH ADI METHOD USING RED-BLACK GAUSS-SIEDEL ALGORITHM FOR IMPLICIT DISCRETIZATION.

```
#include <stdio.h>
#include <stdlib.h>

#define MESHX 100
#define MESHY 100

#define ntime 2000
#define deltat 0.1
#define deltax 1.0
#define deltay 1.0
#define saveT 200
#define radius 20
double c_old[MESHX+2][MESHY+2];

void initialize();
void writetofile (FILE *fp);

void main ()
{
	long t;
	long x,y;
	char filename [1000];

	double a_x[MESHX+1];
	double b_x[MESHX+1];
```

```
	double c_x[MESHX+1];
	double d_x[MESHX+1];

	double a_y[MESHY+1];
	double b_y[MESHY+1];
	double c_y[MESHY+1];
	double d_y[MESHY+1];
	double inv_deltax2 = 1/(deltax*deltax);
	long iter;

	double alpha = inv_deltax2*deltat;

	FILE *fp;
	c_old[MESHX+2][MESHY+2];

	initialize();

	for (t=0;t<ntime;t++)
	{
		//implicit in x using red-black Gauss-Siedel method
		for (y=1;y<MESHY+1;y++)
		{
			//create matrix
			d_x[1] = c_old[1][y] + alpha*c_old[0][y] + alpha*(c_old[1][y-1] -
2.0*c_old[1][y] + c_old[1][y+1]);
			a_x[1] = 0.0;
			b_x[1] = 1.0+2.0*alpha;
			c_x[1] = -alpha;
			for (x=2;x<MESHX+1;x++)
			{
				a_x[x] = -alpha;
				b_x[x] = 1.0+2.0*alpha;
				c_x[x] = -alpha;
				d_x[x] = c_old[x][y] + alpha*(c_old[x][y-1] - 2.0*c_old[x][y] +
c_old[x][y+1]);
			}
			c_x[MESHX]  = 0.0;
			d_x[MESHX] = c_old[MESHX][y] + alpha*c_old[MESHX+1][y] +
alpha*(c_old[MESHX][y-1] - 2.0*c_old[MESHX][y] + c_old[MESHX][y+1]);
			iter=0;
			//ODD x UPDATE using latest updated values of even x
			for(iter=0;iter<=10;iter++)
			{
				for(x=1;x<=MESHX; x+=2)
				{
					c_old[x][y] = (d_x[x] - a_x[x]*c_old[x-1][y] -
c_x[x]*c_old[x+1][y])/b_x[x];
				}
			//EVEN x UPDATE using latest updated values of odd x
				for(x=2;x<=MESHX; x+=2)
				{
					c_old[x][y] = (d_x[x] - a_x[x]*c_old[x-1][y] -
```

```
c_x[x]*c_old[x+1][y])/b_x[x];
                                }
                        }
                }



                //implicit in y using red-black Gauss-Siedel method
                for (x=1;x<MESHX+1;x++)
                        {
                                //create matrix
                                d_y[1] = c_old[x][1] + alpha*c_old[x][0] + alpha*(c_old[x-1][1] -
2.0*c_old[x][1] + c_old[x+1][1]);
                                a_y[1] = 0.0;
                                b_y[1] = 1.0+2.0*alpha;
                                c_y[1] = -alpha;
                                for (y=2;y<MESHY+1;y++)
                                {
                                        a_y[y] = -alpha;
                                        b_y[y] = 1.0+2.0*alpha;
                                        c_y[y] = -alpha;
                                        d_y[y] = c_old[x][y] + alpha*(c_old[x-1][y] - 2.0*c_old[x][y] +
c_old[x+1][y]);
                                }
                                c_y[MESHY]  = 0.0;
                                d_y[MESHY] = c_old[x][MESHY] + alpha*c_old[x][MESHY+1] +
alpha*(c_old[x-1][MESHY] - 2.0*c_old[x][MESHY] + c_old[x+1][MESHY]);
                                iter=0;
                                //ODD y UPDATE using latest updated values of even y
                                for(iter=0;iter<=10;iter++)
                                {
                                        for(y=1;y<=MESHY; y+=2)
                                        {
                                                c_old[x][y] = (d_y[y] - a_y[y]*c_old[x][y-1] -
c_y[y]*c_old[x][y+1])/b_y[y];
                                        }
                                //EVEN y UPDATE using latest updated values of odd y
                                        for(y=2;y<=MESHX; y+=2)
                                        {
                                                c_old[x][y] = (d_y[y] - a_y[y]*c_old[x][y-1] -
c_y[y]*c_old[x][y+1])/b_y[y];
                                        }
                                }
                        }
                        if (t%saveT == 0)
                        {
                                sprintf(filename,"composition%ld.dat", t);
                                fp=fopen (filename,"w");
                                writetofile (fp);
                                fclose (fp);
                        }
                }
```

```
}


void initialize()
{
	long x,y;
	for (x=0;x<=(MESHX+1);x++)
	{
		for(y=0;y<=(MESHY+1);y++)
		{
			if (((x-(MESHX+1)/2)*(x-(MESHX+1)/2) + (y-(MESHY+1)/2)*(y-
(MESHY+1)/2)) < radius*radius)
				c_old[x][y] = 0.8;
			else
				c_old[x][y] = 0.1;
		}
	}
}

void writetofile (FILE *fp)
{
 long x,y;
 for (x=0;x<=(MESHX+1);x++)
 {
	for (y=0;y<=(MESHY+1);y++)
	{
		fprintf(fp, "%le %le %le\n", x*deltax, y*deltay, c_old[x][y]);
	}
	fprintf(fp,"\n");
 }
}
```

2. COMPARISON OF THE NATURE OF THE CONVERGENCE OF THE TWO ALGORITHMS

1. At a particular node (x=45,y=25), composition values were analyzed for successive number of iterations of each algorithm and the difference "Diff" between the compositions at that node for successive number of allowed iterations after 1000 timesteps were observed. The results are plotted in Figure 1 as shown below.
   Diff = (compostion for n iterations) – (composition for n-1 iterations)
   It can be seen that the convergence is reached for both cases at about 20 iterations.

a b

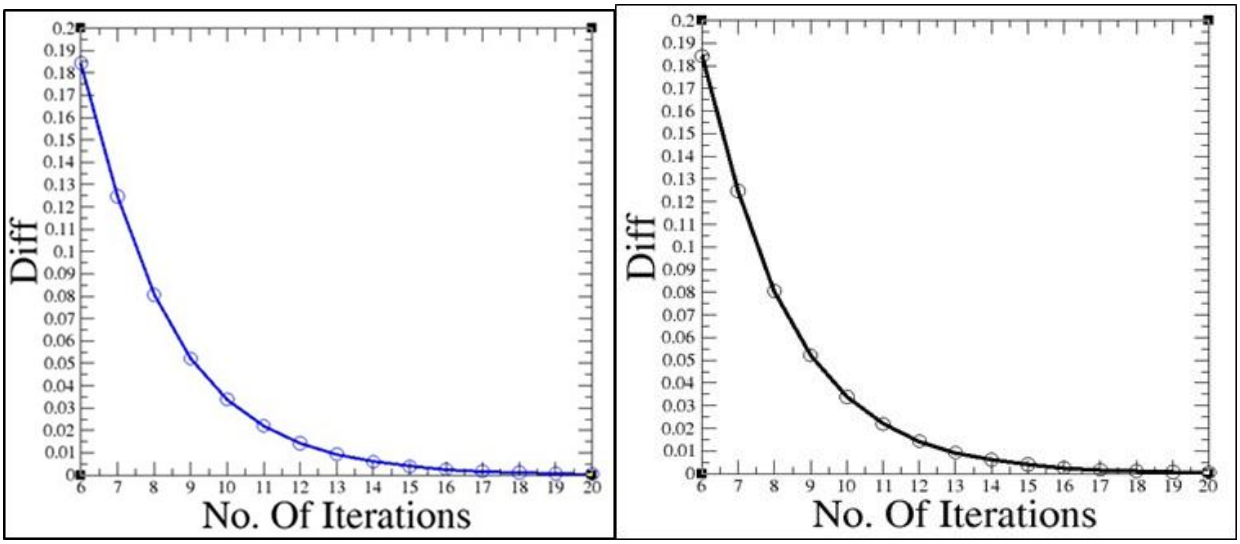

Figure 1 – Convergence of results for (a) Gauss Siedel Algorithm, (b) Jacobi Algorithm

2. From Figure 1, it can be seen that there is not much difference in the nature of the convergence exhibited by the two algorithms.

3. However, it is observed that the time taken by the Red-Black Gauss-Siedel (G-S) algortihm to reach the same level of convergence as the Jacobi algorithm is much faster than the latter. From the CPU execution times of the two programs, it was observed that the program for analysing the convergence of the Jacobi algorithm takes 1779.33 seconds to complete while the same for the G-S algortihm takes 30.84 seconds to complete, indicating that for the same level of convergence, the latter is a much faster algorithm.

3. EFFECT OF CHANGING THE TIME STEP.

It was found that both the Jacobi and the red-black Gauss Siedel algorithms are stable at a maximum time-step (deltat) size of 0.6. Increasing the time step above this value gives unstable results for both algorithms. An example of this is shown in Figure 2 below for 1600 timesteps with both algorithms.

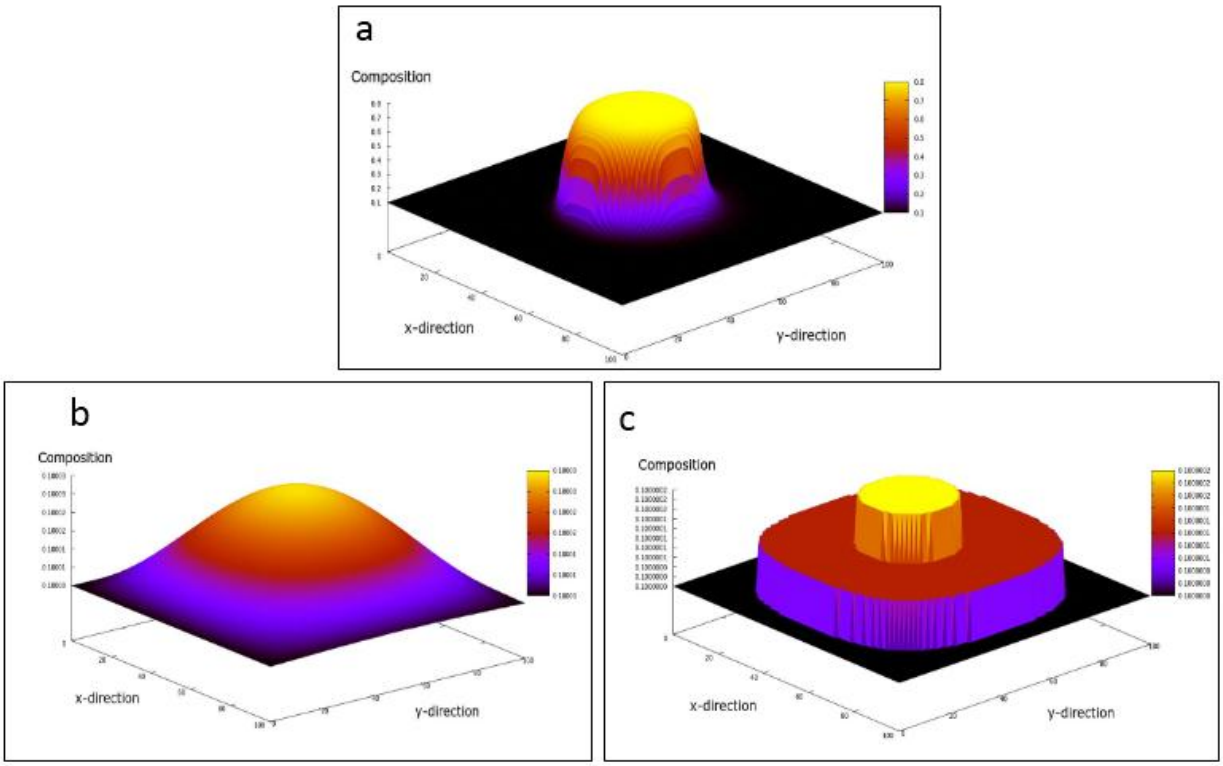

Figure 2 : (a) Intial composition profile. (b) Composition profiles after 1600 timesteps with stepsize below or equal to 0.6. (c) Composition profile at 1600 timesteps with a step size of 0.7. (Same results appear for both algorithms).