Solution to the 1D Heat Equation

Advertisement No. CDS/KA/SERB-SRG/DEC2020/RA

A report submitted for the application of
Project Assistant Position

Sundaresan G
Scientist/Engineer
ISRO Propulsion Complex

B.Tech Mechanical Engineering
Batch of 2016, NITT

Contents

a. Analytical Solution

Given

$$u_t = u_{xx}\ ,\ \ x \in [0, 1] \tag{1}$$

subject to periodic boundary conditions

$$u(0, t) = u(1, t) \quad \text{and} \quad u_x(0, t) = u_x(1, t)$$

and initial condition

$$u(x, 0) = \sin(2\pi x). \tag{2}$$

The given problem is a well posed homogeneous linear PDE and has a unique solution for the given boundary and initial conditions. The solution is derived as follows by separation of variables:

Let

$$u(x, t) = X(x)\,T(t). \tag{3}$$

Substituting into (1), we have

$$XT_t = X_{xx}T \implies \frac{X_{xx}}{X} = \frac{T_t}{T} = \alpha,$$

where α is a constant since the LHS is only a function of x and RHS is a function of t. We now examine three cases for α:

Case 1: $\alpha = 0$

$$X(x) = ax + b \ , \ T(t) = c,$$
$$\implies u^{\alpha=0} = a_1 x + a_2. \tag{4}$$

Case 2: $\alpha = \beta^2 > 0$

$$X(x) = ae^{\beta x} + be^{-\beta x} \ , \ T(t) = ce^{\beta^2 t^2},$$
$$\implies u^{\alpha>0} = (b_1 e^{\beta x} + b_2 e^{-\beta x})e^{\beta^2 t^2}, \tag{5}$$

where b_1 and b_2 are arbitrary constants depending on β.

Case 3: $\alpha = -\lambda^2 < 0$

$$X(x) = a\cos(\lambda x) + b\sin(\lambda x) \ , \ T(t) = ce^{-\lambda^2 t^2},$$
$$\implies u^{\alpha<0} = (c_1 \cos(\lambda x) + c_2 \sin(\lambda x))e^{-\lambda^2 t^2}. \tag{6}$$

where c_1 and c_2 are arbitrary constants depending on λ.

Due to the homogenous and linear nature of the given PDE, a finite sum of (4), (5) and (6) satisfies the PDE. It can be also proved that an infinite convergent sum consisting of (4), (5) and (6) satisfies the PDE. However we would not delve into this topic. For the given problem with the given initial condition, it is enough to consider case 3. Hence,

$$u(x,t) = (c_1 \cos(\lambda x) + c_2 \sin(\lambda x)) \ e^{-\lambda^2 t^2}. \tag{7}$$

Applying boundary condition $u(0,t) = u(1,t)$, we get

$$c_1(\cos(\lambda) - 1) + c_2 \sin(\lambda) = 0 \tag{8}$$

Applying boundary condition $u_x(0,t) = u_x(1,t)$ and the fact that $\lambda \neq 0$, we get

$$c_1 \sin(\lambda) + c_2(1 - \cos(\lambda)) = 0 \tag{9}$$

Solving equations (8) and (9), we get the following two equations

$$c_1 = c_2 = 0.$$

The above equation leads to a trivial solution and does not satisfy the given initial condition. Hence the following must hold

$$\sin(\lambda) = (1 - \cos(\lambda)) = 0,$$

which is satisfied if and only if

$$\lambda = n\pi, \quad \text{where} \quad n \in \mathbb{Z} \setminus \{0\}$$

For the given initial condition (2), we get

$$c_1 = 0, \quad \lambda = 2\pi \quad \text{and} \quad c_2 = 1.$$

The analytical solution is therefore given by

$$u(x,t) = \sin(2\pi x)\, e^{-4\pi^2 t^2} \tag{10}$$

The above solution satisfies the PDE, initial and boundary conditions. Moreover due to well posed nature of the problem, the solution obtained is a unique one (by maximum energy principle).

b. Numerical Scheme

Discretizing (1) by the explicit Euler (time derivative) and second order central difference (space derivative) scheme, we have

$$u_i^{t+\Delta t} = u_i^t + \frac{\Delta t}{(\Delta x)^2}\left(u_{i+1}^t - 2u_i^t + u_{i-1}^t\right), \quad \text{where} \quad i = 1, \ldots, n-1.$$

For $i = 0, n$, the discretized equations, with periodic boundary conditions, yield

$$u_0^{t+\Delta t} = u_0^t + \frac{\Delta t}{(\Delta x)^2}\left(u_1^t - 2u_0^t + u_{n-1}^t\right) \quad \text{and} \quad u_n^{t+1} = u_0^{t+1}.$$

The convergence and stability criteria,

$$\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2},$$

requires $n \leq 223$ for $\Delta t = 0.00001$.

c. Numerical simulations

A code is written in C language and given in Appendix A. It uses command line arguments for input of number of grid points (n) and time step at which solution is required. Please compile it using a suitable compiler (GCC) as follows

gcc code.c -o ”output file name”
”output file name”.exe ”n” ”Time step(s)”

For example,

gcc code.c -o results
results.exe 200 300 500 1000

The above example inputs 200 as the number of grid points and prints the numerical and analytical solutions at 300, 500 and 1000 time steps.

Note: The maximum number of grid points (default:128) the code accepts is 220 (for convergence) and the maximum time step (default: 0, 100, 500 and 1000) is 3000 and number of time step arguments (default:4) is 10.

d. Solution plot for 128 grid points

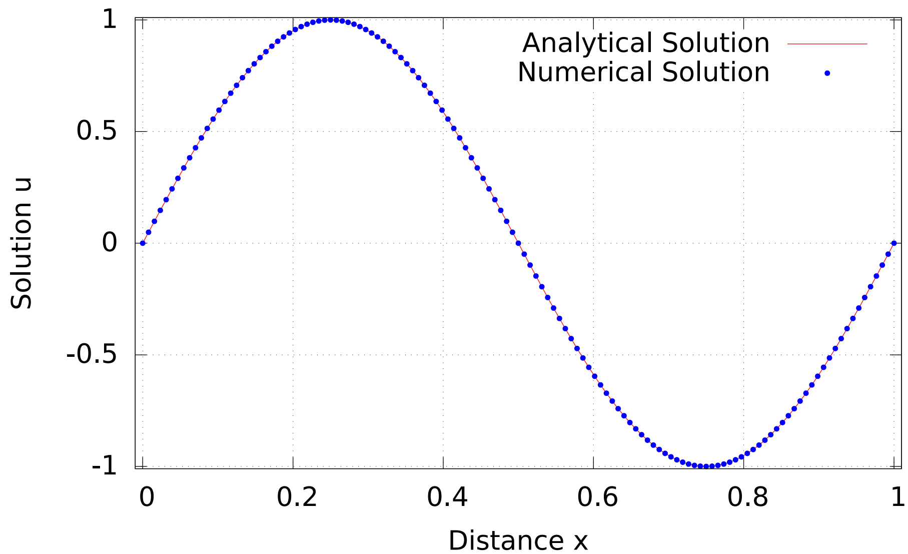

Figure 1: Analytical Vs Numerical Solution at Time step 0.

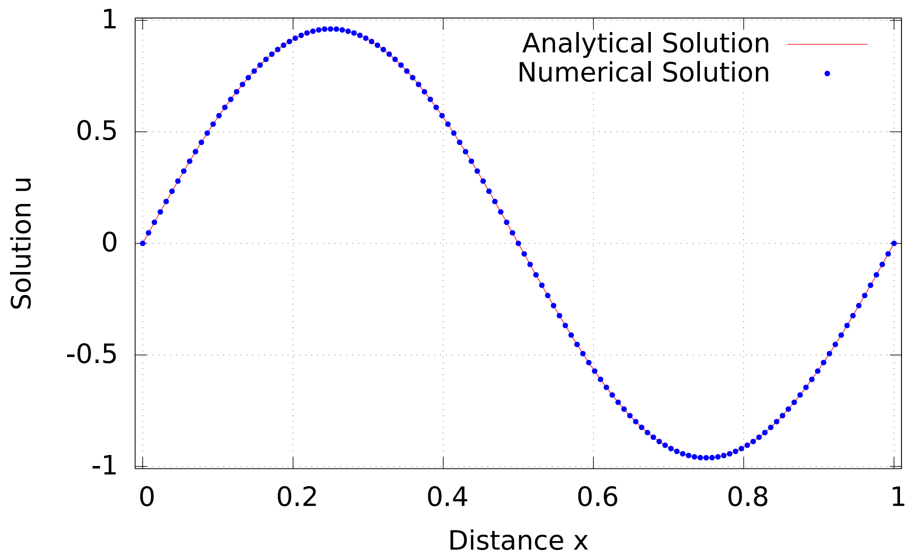

Figure 2: Analytical Vs Numerical Solution at Time step 100.

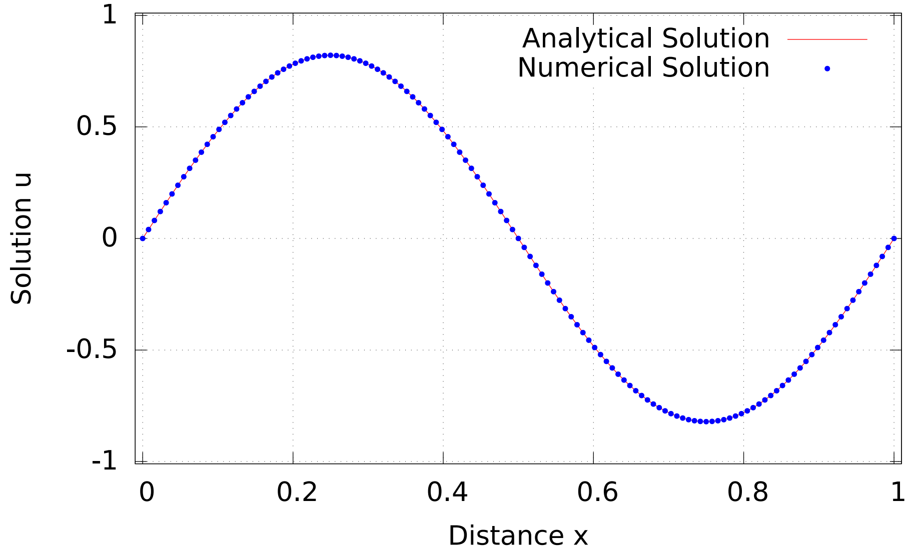

Figure 3: Analytical Vs Numerical Solution at Time step 500.

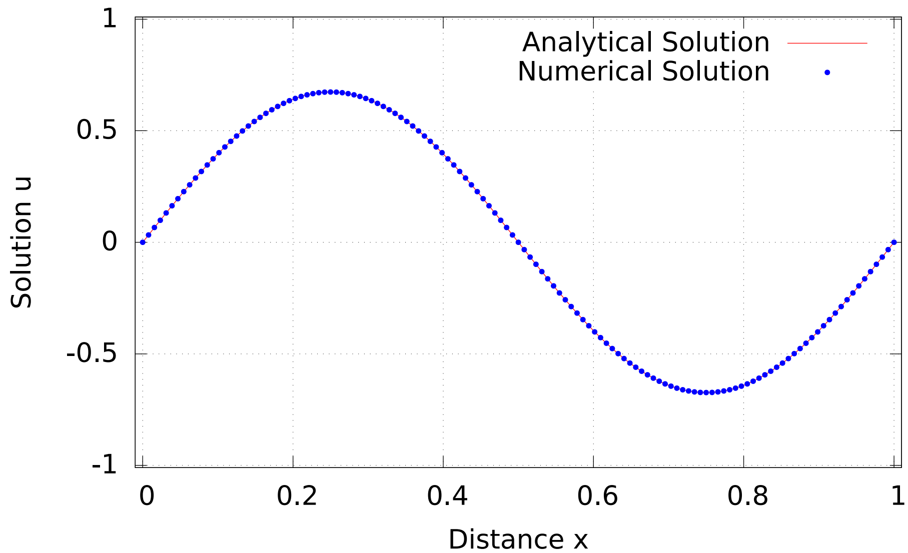

Figure 4: Analytical Vs Numerical Solution at Time step 1000.

e. Average error in the domain

We define the average error to be

$$e_{av} = \sum_i |u_i - u_{ex,i}|\,, \tag{11}$$

where u_i and $u_{ex,i}$ are the numerical and exact solutions respectively at the ith grid point.

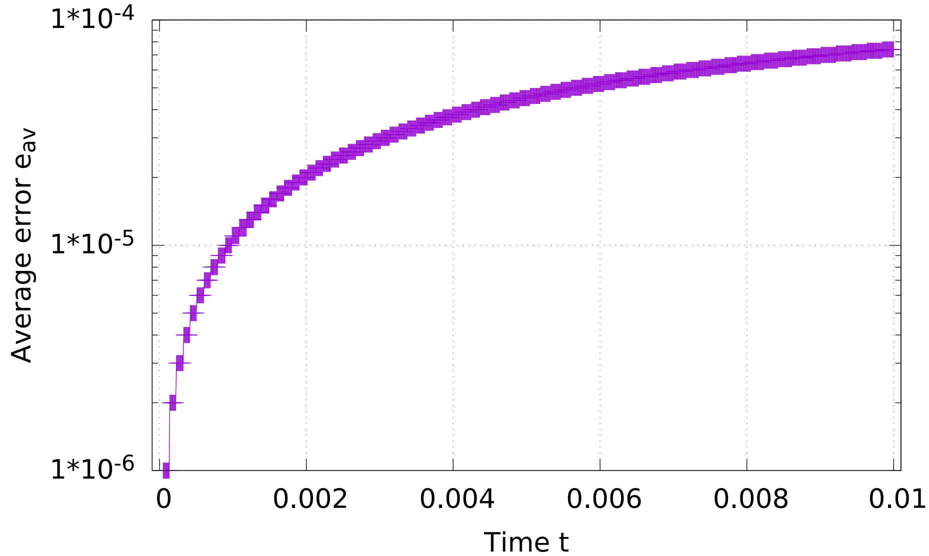

Figure 5: Average error versus time for $n = 128$.

A Appendix A: C code

```c
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
const double pi=acos(-1.0);
int max(int *h, int rows)
{
    int s=0;
    for(int i=0;i<rows;i++)
    {
        if(s<h[i])
            s=h[i];
    }
    return s;
}
void analytical_method(double *as, double delta_t, int grid_points, int time_step, double k, double interval_length)
{
    for(int i=0;i<grid_points;i++)
        as[i]=sin(2*pi*i*(interval_length/(grid_points-1)))*exp(pi*pi*(-4)*k*(time_step)*delta_t);
}
void print_to_file(double *array,int x_points, double delta_x, char *s)
{
    FILE *fptr=fopen(s,"w");
    fprintf(fptr,"# %s\n", s);
    for(int i=0;i<x_points;i++)
        fprintf(fptr,"%lf \t %lf\n", i*delta_x, array[i]);
    fclose(fptr);
}
void error_method(double *error_array, double *exact_solution, double *num_solution, int grid_points)
{
    for(int i=0;i<grid_points;i++)
        error_array[i]=fabs(exact_solution[i]-num_solution[i]);
}
double error_sum_method(double *error_array, int grid_points)
{
    double error_sum=0;
    for(int i=0;i<grid_points;i++)
    {
        error_sum+=error_array[i];
    }
    return error_sum;
}
void initialization(double *array, int grid_points, double interval_length)
{
    for(int i=0; i<grid_points;i++)
        array[i]=sin(2*pi*i*(interval_length/(grid_points-1)));
}
void numerical_solution(double *u, double *uxx, int grid_points, double delta_t, double interval_length, double k)
{
    //array[i+1][columns-1]=array[i+1][0];
    for(int j=0;j<grid_points;j++)
    {
        u[j]+=k*delta_t*uxx[j];
    }
}
void uxx_calculate(double *uxx, double *u, int grid_points, double interval_length)
{
    uxx[0]= (u[1]-2*u[0]+u[grid_points-2])*(grid_points-1)*(grid_points-1)/(interval_length*interval_length);
    uxx[grid_points-1]=uxx[0];
    for(int i=1; i<grid_points-1;i++)
        uxx[i] = (u[i+1]-2*u[i]+u[i-1])*(grid_points-1)*(grid_points-1)/(interval_length*interval_length);
}
int main(int argc, char *argv[])      //argv[1] = number of grid points, argv[i] = numerical and analytical solution printing time step
{
```

```
    int grid_points=128;
    int tc=1000;     //time counter parameter. default of 1000
    double delta_t=0.00001;
    double interval_length=1;
    double k=1;
    int print_time_step[20];

    print_time_step[0]=0;
    print_time_step[1]=100;
    print_time_step[2]=500;
    print_time_step[3]=1000;
    int pts_length=4;               //print_time_step array length. Default is 3+1

    if(argc>1)                  //for intializing printing time steps and grid points based on command line inputs
    {
        grid_points=atoi(argv[1]);
        if(argc>2)
        {
            for(int i=2; i<argc;i++)
            {
                print_time_step[i-2]=atoi(argv[i]);
                if(print_time_step[i-2]==0 || print_time_step[i-2]>3000)
                {
                    printf("\n Invalid time steps for printing. Please specify an integer between 1 and 3000 for every printing time step");
                    return -1;
                }
            }
            tc=max(print_time_step, argc-2);
            pts_length=argc-2;
        }
    }



    if(grid_points==0 || grid_points>220)       //the explicit scheme for the given problem with delta_t=0.00001 converges only if the grid points are less than 223 points
    {
        printf("\n Invalid number of grid points. Please specify number of grid points below 1000");
        return -1;
    }

    grid_points+=1;    //to get even number of points since first and last point are one and same

    //double **analytical_solution= (double **)malloc(time_steps * sizeof(double *));
    /*double **num_solution= (double **)malloc(time_steps*sizeof(double *));
    double **error_array= (double **)malloc(time_steps*sizeof(double *));
    for(int i=0;i<time_steps;i++)
    {
        analytical_solution[i]=(double *)malloc(time_steps*sizeof(double));
        num_solution[i]=(double *)malloc(time_steps*sizeof(double));
        error_array[i]=(double *)malloc(time_steps*sizeof(double));
    }*/

    double *u= (double *)malloc (grid_points*sizeof(double));  //solution
    double *uxx= (double *)malloc (grid_points*sizeof(double));  //second derivative with respect to space
    double *as= (double *)malloc (grid_points*sizeof(double));   //analytical solution
    double *error=(double *)malloc(grid_points*sizeof(double));  //for each grid points
    double *error_sum= (double *)malloc((tc+1)*sizeof(double));     //for sum of errors of each grid points at various time steps

    //printf("\n%p",&analytical_solution[0][0]);
    //initialization(analytical_solution[0], grid_points, interval_length);
    //analytical_method(analytical_solution, delta_t, time_steps, grid_points, k);
    //printf("\n%lf", analytical_solution[100][32]);

    initialization(u, grid_points, interval_length);
```

```c
    analytical_method(as, delta_t, grid_points, 0, k, interval_length);  //to initialize analytical solution at 0

    //printf("\n%lf",u[33]);
    error_sum[0]=0;
    for(int j=0;j<pts_length;j++)
        if(print_time_step[j]==0)   //to dump the analytical and numerical solution at 0th time step.
            {
                char Numsol[1000], Asol[1000];
                sprintf(Numsol,"Numerical solution at %dth time step for %d grid points.txt", 0, grid_points-1);
                print_to_file(u, grid_points, (double)interval_length/(grid_points-1),Numsol);
                sprintf(Asol,"Analytical solution at %dth time step for %d grid points.txt", 0, grid_points-1);
                print_to_file(as, grid_points, (double)interval_length/(grid_points-1),Asol);
            }

    for(int i=1;i<=tc;i++)
    {
        uxx_calculate(uxx, u, grid_points, interval_length);
        /*if(i==1)
            printf("\n%lf",uxx[32]);*/
        numerical_solution(u, uxx, grid_points, delta_t, interval_length, k); //updates u to the next time step
        /*if(i==100)
            printf("\n%lf",u[32]);*/
        analytical_method(as, delta_t, grid_points, i, k, interval_length);
        /*if(i==100)
            printf("\n%lf",as[32]);*/
        error_method(error, as, u, grid_points);
        /*if(i==1000)
            printf("\n%lf",error[97]);*/
        error_sum[i]= error_sum_method(error, grid_points);
        /*if(i==1000)
            printf("\n%lf",error_sum[grid_points-3]);*/
        for(int j=0;j<pts_length;j++)
            if(print_time_step[j]==i)   //to dump the analytical and numerical solution.
            {
                char Numsol[1000], Asol[1000];
                sprintf(Numsol,"Numerical solution at %dth time step for %d grid points.txt", i, grid_points-1);
                print_to_file(u, grid_points, (double)interval_length/(grid_points-1),Numsol);
                sprintf(Asol,"Analytical solution at %dth time step for %d grid points.txt", i, grid_points-1);
                print_to_file(as, grid_points, (double)interval_length/(grid_points-1),Asol);
            }
    }

    char error_string[]="Sum of errors at each grid point for various time steps.txt";
    print_to_file(error_sum, tc+1, delta_t, error_string);


    free(error_sum);
    free(error);
    free(as);
    free(uxx);
    free(u);
}
```