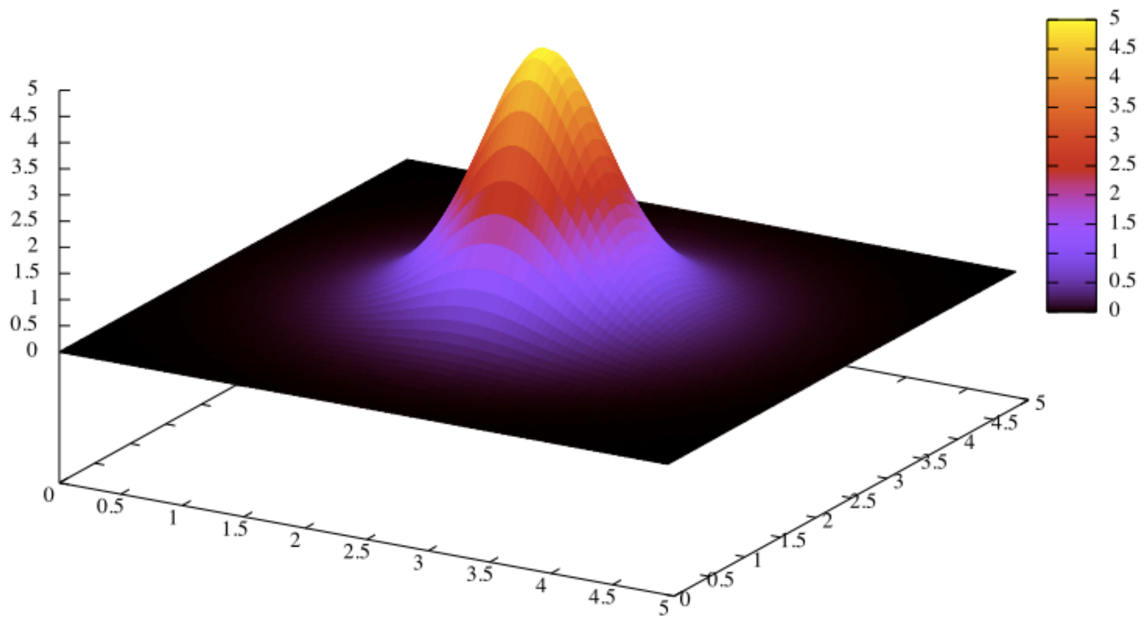
# Numerical Solution for 2D Heat Equation

Xiaobo Sun



## 1. Introduction

Heat equation is a partial differential equation, which describes the distribution of heat in a given region over time. 2D heat equation is:

$$U_t = C ( U_{xx} + U_{yy})$$

In this project, we will use numerical method to solve the equation.

## 2. Version Management

The code of this project is managed using git and github. The repository can be found at the following URL:

https://github.com/sxbwjm/2D-Heat-Equation.git

## 3. Implementation

The project is implemented by C, which includes the following parts:

### 3.1 Discretization in Space (heat_equation.c)

We first discretize the equation in 2D space, so that the computer can calculate the derivatives of x and y.

The boundary is defined as: [0:5] x [0:4]

This is implemented in "heat_equation.c".

There are five functions in this file:

. D1U( u1, u2, delta):
  Calculate the first derivative of x or y location in the middle of u1 and u2.
  It returns (u2 – u1) / delta.

. D2U(u_1, u, u1, delta):
  Calculate the second derivative of x or y location at u, using D1U function.

. rhsFunc(U, newU):
Calculate the values of the right hand side expression for all (x, y) locations. U and

newU are 2D arrays of u values.

. fillGhostCells
Fill the ghost cells for calculation of cells at boundary.

## 3.2 Timestepping (Runge-Kutta.c)

Runge Kutta algorithm is a method of numerically integrating ordinary differential equations.

Given equation: u' = f(t, u) with initial condition: u(0) = u0. Suppose that $u_n$ is the value of the variable at time $t_{n.}$ The Runge-Kutta formula takes $u_n$ and $t_n$ and calculates an approximation for $u_{n+1}$ at a brief time later, $t_{n+h}$. It uses a weighted average of approximated values of f (t, u) at several times within the interval ($t_{n,}$ $t_n$+h). The 4th order formula (also known as RK4) is given by:

k1 = f($t_n$, $u_n$)
k2 = f($t_n$+ h/2, $u_n$ + k1 * h / 2)
k3 = f($t_n$+ h/2, $u_n$ + k2 * h / 2)
k4 = f($t_n$+ h, $u_n$ + k3 )

$u_{n+1}$ = $u_n$ + (k1 + 2 * k2 + 2 * k3 + k4) * h / 6

Since we have the function (rhsFunc) for calculating the right hand side expression in the equation of:

$$Ut = C(U_{xx} + U_{yy})$$

If the initial value U0 is know at t = 0, it is easy to get U values at any time t using Runge-Kutta 4th formula. It is worth pointing out there is no “t” variable in the right hand side function in the above equation.

The Runge-Kutta algorithm is implemented in “Runge-Kutta.c”.

## 3.3 Initial Conditions (init_conditions.c)

To use Runge-Kutta method to get U values at any time, the initial condition at t = 0 is necessary:

$$Ut0 = f(x, y, 0)$$

Theoretically, f(x, y, 0) can be any function. However, we need to choose some special functions as initial conditions to verify our implementation. Three functions are used:

1) $u(x,y,0) = (\frac{c}{2})/cosh^2\left(\frac{\sqrt{c}}{2}(x-2.5)\right)$ , where c = 5

2) $u(x,y,0) = (\frac{c}{2})/cosh^2\left(\frac{\sqrt{c}}{2}\left(\sqrt{(x-2.5)^2 + (y-2.5)^2}\right)\right)$, where c = 5

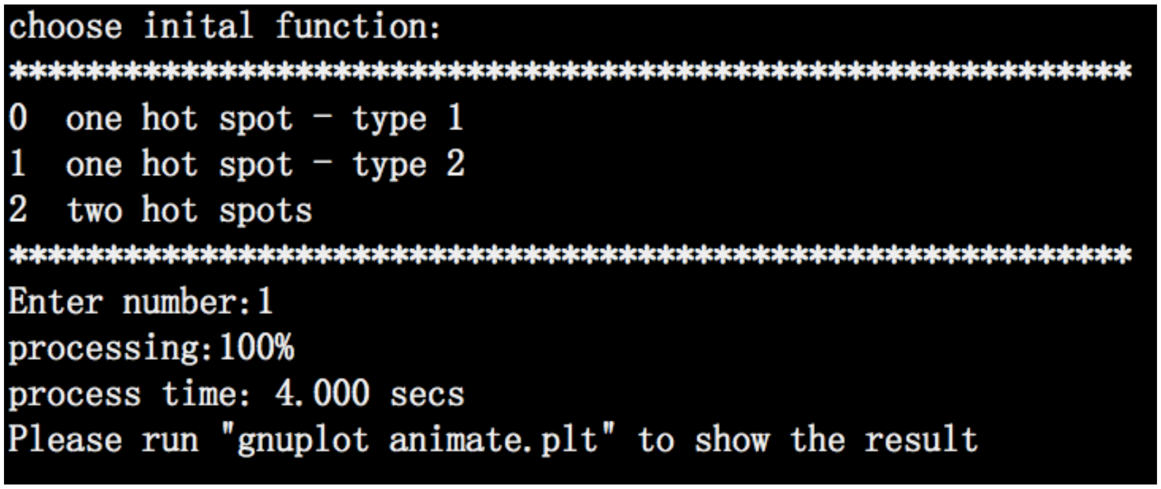
3) $u(x,y,0) = (\frac{c1}{2})/cosh^2\left(\frac{\sqrt{c1}}{2}\left(\sqrt{(x-1.5)^2 + (y-1.5)^2}\right)\right)$ +

$(\frac{c2}{2})/cosh^2\left(\frac{\sqrt{c2}}{2}\left(\sqrt{(x-3.0)^2 + (y-3.0)^2}\right)\right)$

, where c1 = 20, c2 = 10

1) & 2) are one hot spot, 3) is a two hot spots.

In the program, these functions are saved in a function pointer array, and user can choose which function to use by entering a number:

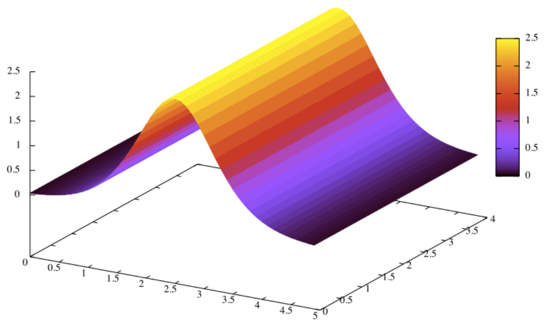


```
choose inital function:
************************************************************
0  one hot spot - type 1
1  one hot spot - type 2
2  two hot spots
************************************************************
Enter number:1
processing:100%
process time: 4.000 secs
Please run "gnuplot animate.plt" to show the result
```

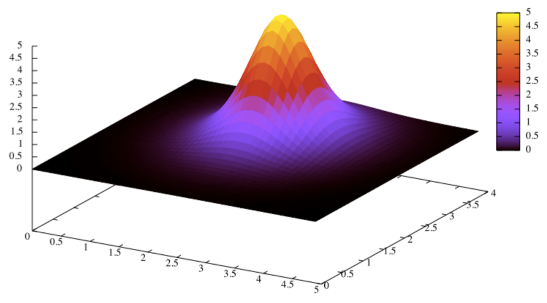
### 3.4 GNUPlot Script (plot_script.c)

(x, y, z) values are saved in text files ( one file for one time frame), and can be plotted as animation in GNUPlot. We use GNUPlot script to iterate all text files to generate the amination. Since different initial function results in different max u values, to get the best animation, it is necessary to specify different range for u in the script. "plot_script.c" generate the script dynamically based on the user's selection on the initial function. After all data is processed, user can run "gnuplot animate.plt" (in the project directory) from the command line to watch the result.

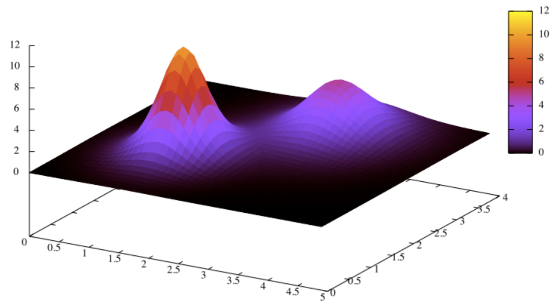
## 3.5 Output



1. $u(x,y,0) = (\frac{c}{2})/cosh^2\left(\frac{\sqrt{c}}{2}(x-2.5)\right)$



2. $u(x,y,0)(\frac{c}{2})/cosh^2\left(\frac{\sqrt{c}}{2}\left(\sqrt{(x-2.5)^2+\ (y-2.5)^2}\right)\right)$



3. $u(x,0) = (\frac{c1}{2})/cosh^2\left(\frac{\sqrt{c1}}{2}\left(\sqrt{(x-1.5)^2+\ (y-1.5)^2}\right)\right) + (\frac{c2}{2})/cosh^2\left(\frac{\sqrt{c2}}{2}\left(\sqrt{(x-3.0)^2+\ (y-3.0)^2}\right)\right)$

graph 1 and graph 2 are one hot spot; graph 3 is two hot spots.

* These output are recorded as gif animate files and saved in “gifs” directory.

## 3.6 Parallel Computation (MPI)

2D Heat equation requires much more calculation than 1D. We use MPI to implement multiple processes computation.

MPI stands for Message Passing Interface, which makes it easier for processes to send and receive messages.

We split the 2D array computation into multiple blocks in Y direction. In the outer for loop, instead of using:

```
for(int y = 0; y < Y_N; y++)
```

We use:

```
for(int y = glbYStart; y < glbYEnd; y++)
```

, where glbYstart and glbYEnd are the start index and end index in Y direction, which is calculated based on the rank of the current process. We measured processing time by changing X steps to 200 and Y steps to 40:

Table 1: processing time (seconds)

| **Process=1** | **Process=2** | **Process=4** | **Process = 8** |
|---|---|---|---|
| 6 | 3 | 2 | 1.5 |

As we can see from the table, increasing the number of processes can dramatically improve the performance.

## 4. References

[1] Ryan C. Daileda, Trinity University, The two dimensional heat equation:
http://ramanujan.math.trinity.edu/rdaileda/teach/s12/m3357/lectures/lecture_3_6_short.pdf

[2] Matthew J. Hancock, The heat and wave equations in 2D and 3D:
http://www.myoops.org/course_material/mit/NR/rdonlyres/61EEDCBD-B969-4771-A183-2831D7420509/0/pde3d.pdf

[3] Verena Horak, Peter Gruber, University of Salzburg, Parallel Numerical Solution of 2-D Heat Equation:
http://www.cosy.sbg.ac.at/events/parnum05/book/horak1.pdf