# Fast Heat Equation Simulator within 2D Rectangle Boundary

Yingqi Bian (Team Eternal Core)

## 1 Overview

The problem of my group is to provide a good and fast algorithm to simulate heat transfer within a rectangle boundary under various settings, such as initial conditions, boundary conditions and etc. By applying PyTorch framework and smart code logic, an relatively accurate result can be obtained under an acceptable amount of time. You can find the code implementation by `https://github.com/yuitoryu/heatsimu`.

## 2 Problem with Analytical solution

A simple boundary value problem (BVP) on heat equation can be in the following from:

$$\begin{cases} u_t = c(u_{xx} + u_{yy}) \\ u(0,y,t) = u(L_x,y,t) = 0 \\ u(x,0,t) = u(x,L_y,t) = 0 \\ u(x,y,0) = \psi(x,y) \end{cases} \tag{1}$$

where $t > 0, (x,y) \in (0,L_x) \times (0,L_y)$. An analytical solution will be:

$$u(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} A_{mn} e^{-c\pi^2\left((\frac{m}{L_x})^2+(\frac{n}{L_y})^2\right)t} \sin(\frac{m\pi}{L_x})\sin(\frac{n\pi}{L_y}) \tag{2}$$

$$A_{mn} = \frac{4}{L_x L_y}\int_0^{L_x}\int_0^{L_y} \psi(x,y)\sin(\frac{m\pi}{L_x})\sin(\frac{n\pi}{L_y})\,dydx \tag{3}$$

Although a completely form of solution can be given, some problem may arise. First, for most time we will get $u(x,y,t)$ in form of an infinite series where one have to approximate it by take partial sum of first finite number of terms. For some series that converge in a slower manner, the cost for an relatively accurate result is high.

Second, unlike differentiation, not all elementary function has an anti-derivative in form of elementary functions. To approximate $A_{mn}$'s, one has to either evaluate the infinite series (this can be done by Taylor expansion) or use Riemann sum to discretize the integral. Under the necessity of both infinite series and its coefficient, it is hard to get an acceptable result under reasonable amount of time.

What's more, problems in real life usually go much more complicated than our simple assumption. Consider the following BVP.

$$\begin{cases} u_t = \frac{\partial}{\partial x}[c(x,y)u_x] + \frac{\partial}{\partial y}[c(x,y)u_y] + Q(x,y,t) \\ a_{left}u(0,y,t) + b_{left}(0,y,t) = \phi_{left}(y,t) \\ a_{right}u(L_x,y,t) + b_{right}(L_x,y,t) = \phi_{right}(y,t) \\ a_{up}u(x,0,t) + b_{up}(x,0,t) = \phi_{up}(x,t) \\ a_{down}u(x,L_y,t) + b_{down}(x,L_y,t) = \phi_{down}(x,t) \\ u(x,y,0) = \psi(x,y) \end{cases} \tag{4}$$

where $c(x,y)$ is location dependent and $Q(x,y,t)$ is the heat source. Unfortunately, there is no closed form solution for this slightly complicated BVP. Thus, we have to find an efficient algorithm to simulate the result. Note that throughout this report, I will be focusing on (4) unless stated.

## 3 Framework and Logic of Code

To attain a good result while keeping convenience for development, I chose PyTorch to build my solver. PyTorch is a fantastic library known for its application in deep learning. In my solver, I focused on utilizing its amazing features on tensor computation including GPU, JIT and etc. The overall logic of the code is like:

**Algorithm 1** Initialization of Simulation

**Require:** Parameters of Heat Equation, Steps
**Ensure:** $\gcd(a, b)$
1: Variable initialization
2: **if** Sanity check not passed **then**
3: Raise error and exit
4: **end if**
5: Further initialization
6: Pick computation mode
7: Waiting for start of simulation

**Algorithm 2** Performing Simulation

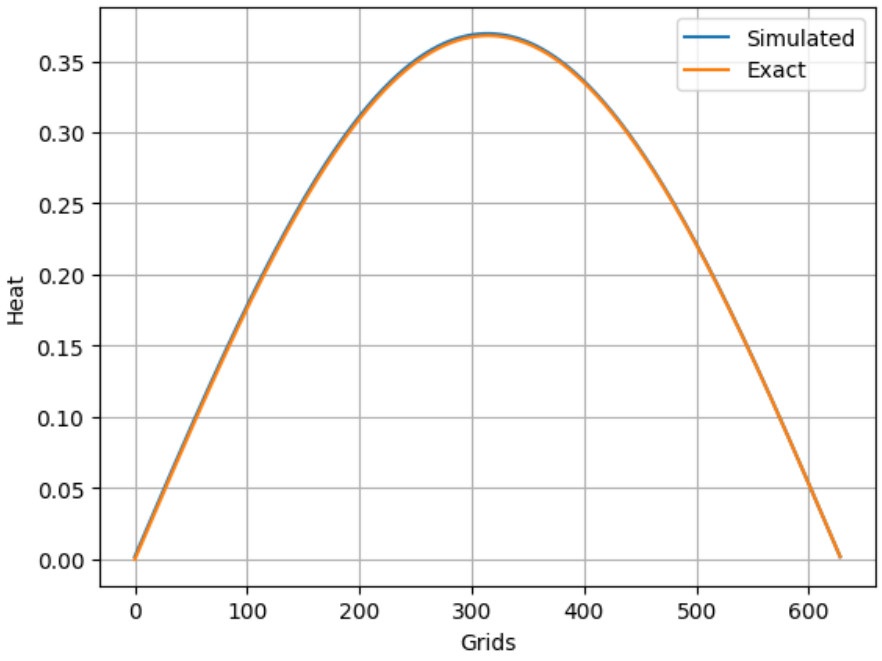
**Require:** None
**Ensure:** Evolution of Heat
1: **for** each time step **do**
2: Set boundary condition
3: Forward by convolution
4: Increment current time
5: **if** time_step mod plot_step == 0 **then**
6: Record grid data
7: Update max and min temperature
8: **end if**
9: **end for**
10: Plot heat transfer

# 4 Evaluation of simulation result

Consider the following BVP.

$$\begin{cases} u_t = u_{xx} + u_{yy} \\ u(0, y, t) = u(\pi, y, t) = 0 \\ u_y(x, 0, t) = u_y(x, \pi, t) = 0 \\ u(x, y, 0) = \sin x \end{cases} \tag{5}$$

which has a simple solution: $u(x, y, t) = e^{-t} \sin x$. This heat equation is extended from a 1D BVP and that's why the solution is independent of $y$. Thus, the evaluation will also ignore $y$ and treat the result as a rod instead of a plane. Here is the simulated and the analytical results.



The time step is $6.25e-6$ and the grid step is 0.005. The error is computed through Frobenius norm it is $5.85e-05$. Thus, I can conclude that my simulator does provide accurate result.

# 5 Detecting Stupid Input

Good simulation requires good parameter settings. Although this cannot be precisely measured, it is for sure that a tremendous amount of time can be saved. Here is a list of sanity checks that are done before allowing any simulation session to start.

- Boundary conditions. Avoid illegal zeros.
- Size of time steps and grid steps. Avoid too big values.
- Stability check. Prevent instability during simulation (or NaN will dominate the grid).

# 6 Benchmark Setting on Optimization

The benchmark is performed on Windows laptop with a i9-14900HX and RTX 4070 laptop GPU. The test case is universal for all benchmark and has following settings:

- $(x, y) \in (0, \pi) \times (0, \pi)$
- time step $= 0.00005\,\mathrm{s}$
- grid step $= 0.05$
- conductivity $= 0.5$
- heat source $Q(x, y, t) = -5\sin(5x)\sin(5y)\cos\sqrt{(x-\frac{\pi}{2})^2+(y-\frac{\pi}{2})^2}\sin(\frac{\pi}{8}t)$
- $\alpha_{left} = \alpha_{right} = \alpha_{up} = \alpha_{down} = 1$
- $\beta_{left} = \beta_{right} = \beta_{up} = \beta_{down} = 0$
- $\phi_{left}(y,t) = \phi_{right}(y,t) = \begin{cases} 0, y \notin [\frac{\pi}{4}, \frac{3\pi}{4}] \\ 1, y \in [\frac{\pi}{4}, \frac{3\pi}{4}] \end{cases}$
- $\phi_{up}(x,t) = \phi_{down}(x,t) = \begin{cases} 0, x \notin [\frac{\pi}{4}, \frac{3\pi}{4}] \\ 1, x \in [\frac{\pi}{4}, \frac{3\pi}{4}] \end{cases}$

The time is measured by %timeit magic line since it gives precise measurement and also standard deviation which can tell whether the experiment result is stabilized or not.

# 7 Optimization on Convolution and by Mode Select

The convolutional forward process has been accelerated with @torch.compile, a Pytorch JIT that focuses on PyTorch native operation. It is picked for optimization since it is one of the most frequent called modules. Here, I propose 3 basic algorithms for handling different type of conuctivity:

- kernel 1: convolution with one simple kernel for constant conductivity
- kernel 2: convolution with 4 split kernels for variable conductivity
- kernel 3: convolution with 4 output channels for for variable conductivity

Here is the result of benchmark on convolutional modules.

| | Native | @torch.compile | Relatively saved time |
|---|---|---|---|
| kernel 1 | $112\ \mu s \pm 4.69\ \mu s$ | $64.6\ \mu s \pm 3.15\ \mu s$ | $42.32\% \pm 5.35\%$ |
| kernel 2 | $145\ \mu s \pm 4.6\ \mu s$ | $69\ \mu s \pm 5.92\ \mu s$ | $52.41\% \pm 5.43\%$ |
| kernel 2 | $187\ \mu s \pm 1.54\ \mu s$ | $116\ \mu s \pm 28.6\ \mu s$ | $37.97\% \pm 15.32\%$ |

The result between native PyTorch and JIT strongly suggests that JIT can provides incredible improvement on running time. On the other hand, the result also indicates the necessity for a selection of computation mode since simulation with constant conductivity runs faster with a simpler kernel.

# 8 Optimization on boundary condition update

Another repeatedly called module during simulation loop if updating boundary condition. Here is 3 sets of results on optimization of this module.

| | Time taken | Relatively saved time |
|---|---|---|
| Native | $659\ \mu s \pm 22.5\ \mu s$ | 0% |
| @torch.compile | $56\ \mu s \pm 3.06\ \mu s$ | $91.5\% \pm 4.65\%$ |
| @torch.compile (fullgraph=True) | $52.1\ \mu s \pm 4.06\ \mu s$ | $92.09\% \pm 4.68\%$ |

The boundary condition modules involves large amount of elementary tensor operations, thus it is not very surprising that a huge improvement is made. Plus, with "fullgraph=True" enabled, I also get a tiny boost in the performance.

# 9 Overview on Optimization All Together

| | Before optimization | After optimization | Relatively saved time |
|---|---|---|---|
| Constant conductivity | $16.6\,s \pm 380\,ms$ | $6.07\,s \pm 75.9\,ms$ | $63.43\% \pm 2.75\%$ |
| Non-constant conductivity | $18.1\,s \pm 297\,ms$ | $6.59\,s \pm 56\,ms$ | $63.59\% \pm 1.97\%$ |

By measuring time taken for complete simulation session, it can be seen that roughly 63.5% can be saved with various amount of time.

# 10 Conclusion

To sum up, my simulator does given accurate simulator result within short amount of time. The optimization focuses on saving time on failed input, good code logic and PyTorch JIT implementation. Detailed check on simulation parameters effectively reduce the chance of obtaining bad simulation result, thus time is saved. The smart choice of computation pattern well handles simpler problems, avoiding additional time cost on using more complicated algorithm for more general problems. For JIT, given that the simulation is performed based on discretization, elementary operations such as addition and multiplication are heavy. Thus, the use of JIT provide a significant boost in the computation speed.