University College London

PHAS0102 - High Performance Computing

Project - Stage 2

---

# Solving Parabolic Diffusion Equation Using OpenCL Parrallelisation

---

*Author:*
Fahad Chohan

*Supervisor:*
Dr. Timo Betcke

March 29, 2019

# 1 Introduction

The second stage of this project aimed to create a diffusion solver from the time dependant parabolic equation (Eq.1)

$$u_t(x,y,t) = \nabla \cdot (\sigma(x,y)\nabla)\, u(x,y,t) \tag{1}$$

with boundary conditions:

- $(x,y) \in [0,1] \times [0,1]$.
- $u(x,y,t) = 0$ on $\Gamma_D$ for $t \geq 0$, $\Gamma_D$ is the values of $u(x,y,t)$ on the edges.
- $u(x,y,t) = g(x,y)$ for $t = 0$, $g(x,y)$ is supplied to the solver.

Similarly to stage 1, the operator $\nabla \cdot (\sigma(x,y)\nabla)\, u(x,y)$ can be approximated by Eq.2

$$\nabla \cdot (\sigma(x,y)\nabla)\, u(x,y) \approx A + B \tag{2}$$

$$A = \frac{\left(\sigma_{i+1/2,j}\frac{(u_{i+1,j}-u_{i,j})}{h}\right) - \left(\sigma_{i-1/2,j}\frac{(u_{i,j}-u_{i-1,j})}{h}\right)}{h}$$

$$B = \frac{\left(\sigma_{i,j+1/2}\frac{(u_{i,j+1}-u_{i,j})}{h}\right) - \left(\sigma_{i,j-1/2}\frac{(u_{i,j}-u_{i,j-1})}{h}\right)}{h}$$

where we approximate $\sigma$ by $\sigma_{i+1/2,j} \approx \frac{1}{2}(\sigma_{i+1,j} + \sigma_{i,j})$ and $h$ is the spacing between matrix elements.

Using a forward time scheme, $u_t(x,y,t)$ can be expressed as in Eq.3 and by substituting Eq.1 we obtain Eq.4.

$$u_t(x,y,t) \approx \frac{u(x,y,t+\Delta t) - u(x,y,t)}{\Delta t} \tag{3}$$

$$u(x,y,t+\Delta t) \approx [1 + \Delta t\ \nabla \cdot (\sigma(x,y)\nabla)]u(x,y,t) \tag{4}$$

There is a limit on the maximum usable $\Delta t$ before the values for $u(x,y,t)$ blow up. We can find this maximum from our forward time scheme relationship, Eq.4, where we have eigenvalues $1 + \Delta t \lambda_k$ where $\lambda_k$ are the eigenvalues for the operator $\nabla \cdot (\sigma(x,y)\nabla)$. Thus, we require that $|1 + \Delta t \lambda_k| < 1$ for all eigenvalues. From this one determines that we must have Eq.5, where $\lambda_{km}$ is size of the largest eigenvalue.

$$\lambda_{km} < \frac{2}{\Delta t} \tag{5}$$

# 2 Code Implementation

The first part of the code imports relevant modules and creates a `Timer()` and `variable()` class which will be used to time implementations and which store certain variable values.

The next part of the code has the openCL implementation of the matrix vector multiplication approximation from Eq.2. This is the same as in stage 1 with the exception of a minus sign. There is also a visual representation of the $g(x,y)$ and $\sigma(x,y)$ matrices.

The openCL matrix vector function takes a vector $u$ (flattened $u(x, y)$ using `ravel()` function) as a parameter and applies Eq.2 to all elements in a matrix free way to return a vector $Au$. Boundary conditions are preserved by an if statement in the kernel.

The openCL implementation uses a work-group size of 1 and applies the matrix operation on all the elements simultaneously as they do not need to be synchronised within each solver iteration. When the iterative solvers apply the next iteration all the values of the vector are up to date, ensured by the `queue.finish()` command.

The final part of the code starts with creating an `nIter` variable which is used to determine the number of time iterations to perform, and a `DATA[]` list which stores $u(x, y, t)$ values after each time iteration. Next, four python functions are created, `max_timestep()`, `do_timestep()`, `iteration_loop()` and `show_iterations()`. The first of these calculates the maximum usable $\Delta t$ value as according to Eq.5. The `do_timestep()` function takes in values for $u(x, y, t)$ and outputs $u(x, y, t + \Delta t)$ accordingly with Eq.4. The `iteration_loop()` function uses a for loop to find a certain number of time iterations for $u(x, y, t)$ and stores them in the `DATA[]` list. The final of these functions, `show_iterations()`, prints visual representations of the $u(x, y, t)$ values as well as their timestamp and iteration number.

The code then uses a $\Delta t$ value at 90% of the calculated maximum and uses the python functions to perform time iterations and print visuals to screen. Additionally, `Axes3D` can be imported from `mpl_toolkits.mplot3d.axes3d` to create 3d visuals as seen below but this has been left out the code for time convenience.

# 3 Experiments and Results

Six different starting conditions, $g(x, y)$ values, and diffusion results are shown in Fig.1-6. For these simulations, 4000 iterations were performed with $300 \times 300$ element $u(x, y, t)$ matrices with a $\Delta t = 2.43$ $\mu$s. 4000 iterations took $\approx 30$ seconds (10 iterations took $\approx 70$ ms).

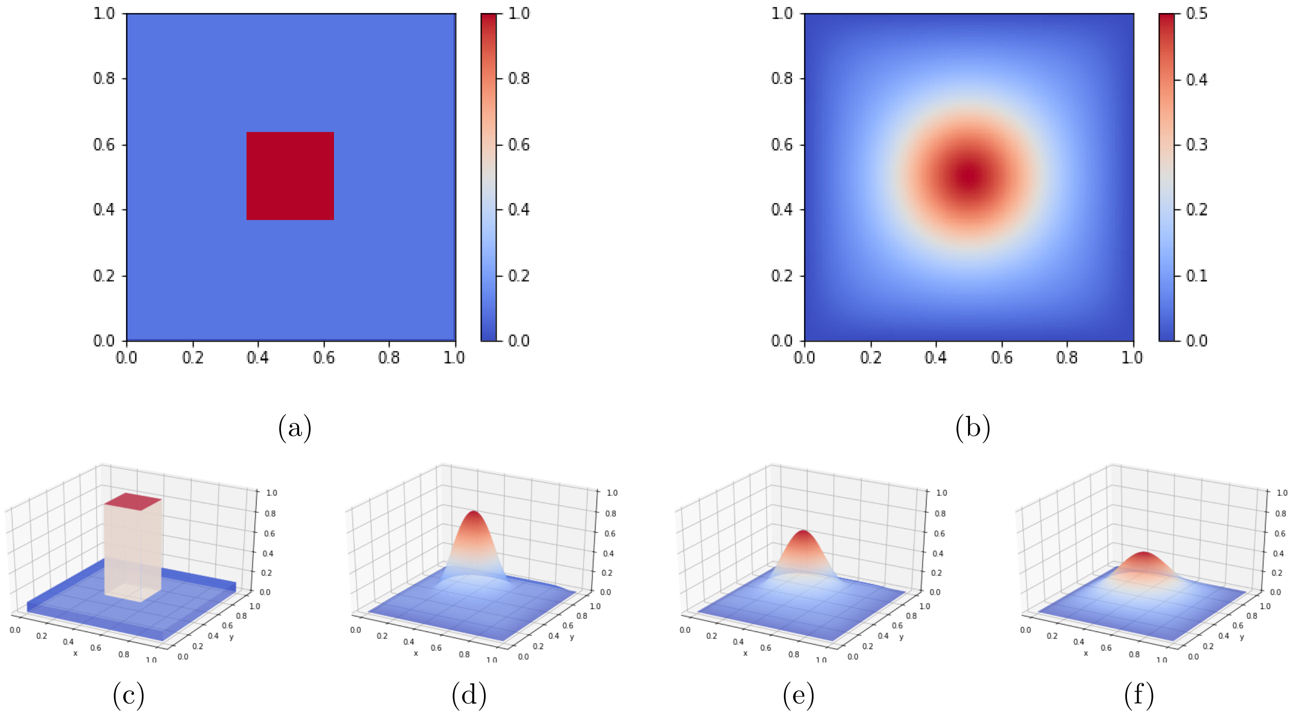


Figure 1: Box diffusion. (a) Start at $t = 0$ ms. (b) Finish at $t = 4.71$ ms with 4000 iterations. (c-f) 3D visual of $u(x, y, t)$ at $t = 0$ ms, 2.35 ms, 4.71 ms, 9.41 ms and iteration count 0, 1000, 2000 and 4000 respectively.

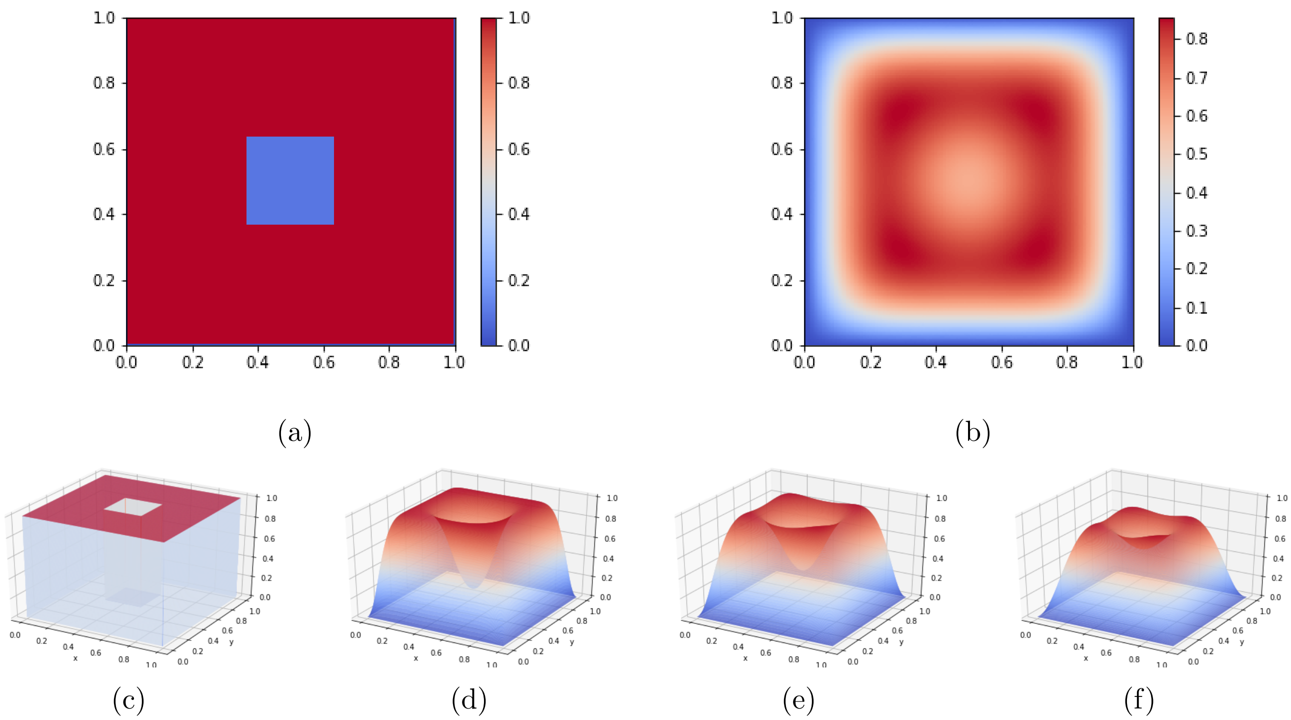


Figure 2: Hollow box diffusion. (a) Start at $t = 0$ ms. (b) Finish at $t = 4.71$ ms with 4000 iterations. (c-f) 3D visual of $u(x, y, t)$ at $t = 0$ ms, 2.35 ms, 4.71 ms, 9.41 ms and iteration count 0, 1000, 2000 and 4000 respectively.

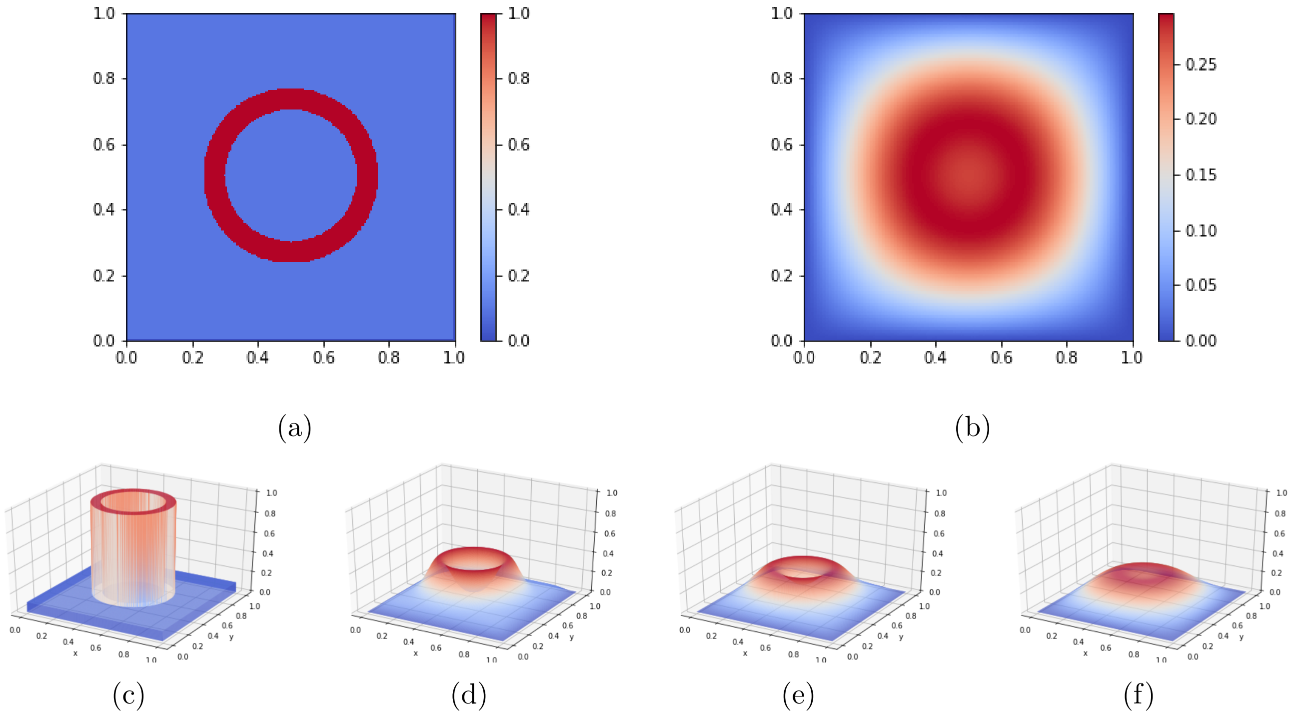

Figure 3: Ring diffusion. (a) Start at $t = 0$ ms. (b) Finish at $t = 4.71$ ms with 4000 iterations. (c-f) 3D visual of $u(x, y, t)$ at $t = 0$ ms, 2.35 ms, 4.71 ms, 9.41 ms and iteration count 0, 1000, 2000 and 4000 respectively.

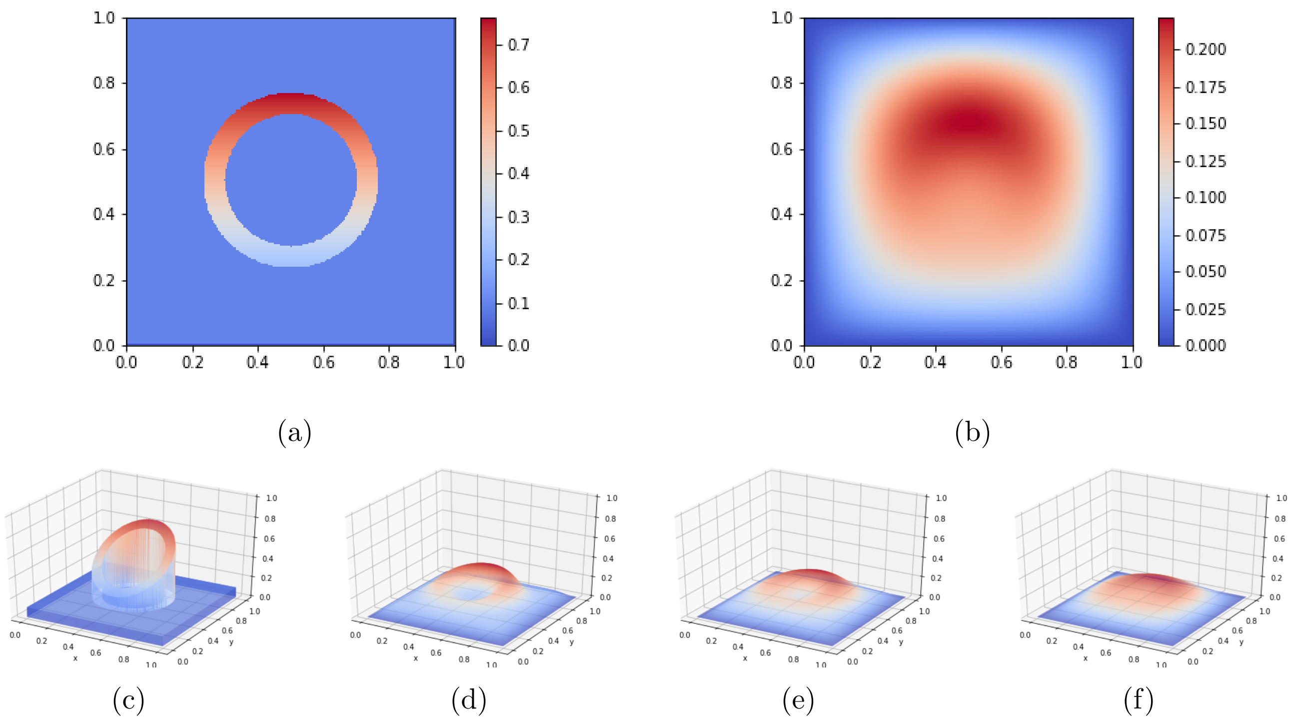

Figure 4: Tilted ring diffusion. (a) Start at $t = 0$ ms. (b) Finish at $t = 4.71$ ms with 4000 iterations. (c-f) 3D visual of $u(x, y, t)$ at $t = 0$ ms, 2.35 ms, 4.71 ms, 9.41 ms and iteration count 0, 1000, 2000 and 4000 respectively.

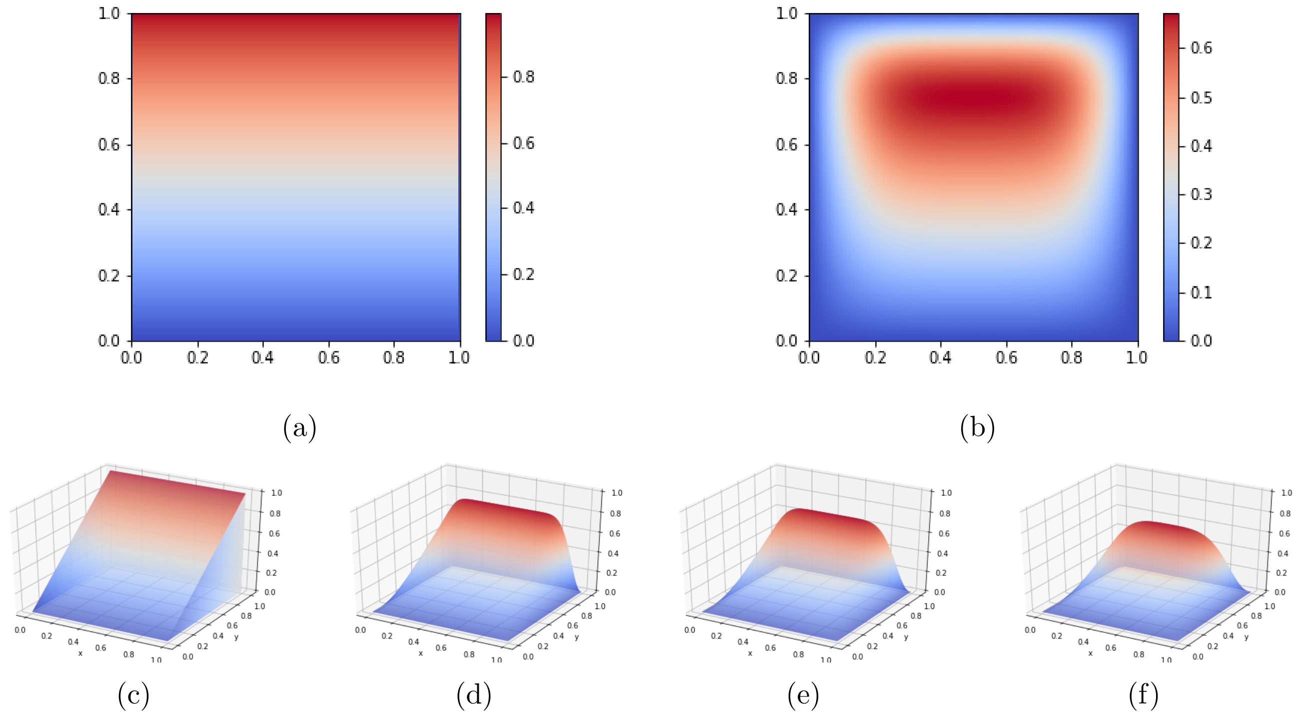


Figure 5: Doorstop diffusion. (a) Start at $t = 0$ ms. (b) Finish at $t = 4.71$ ms with 4000 iterations. (c-f) 3D visual of $u(x, y, t)$ at $t = 0$ ms, 2.35 ms, 4.71 ms, 9.41 ms and iteration count 0, 1000, 2000 and 4000 respectively.

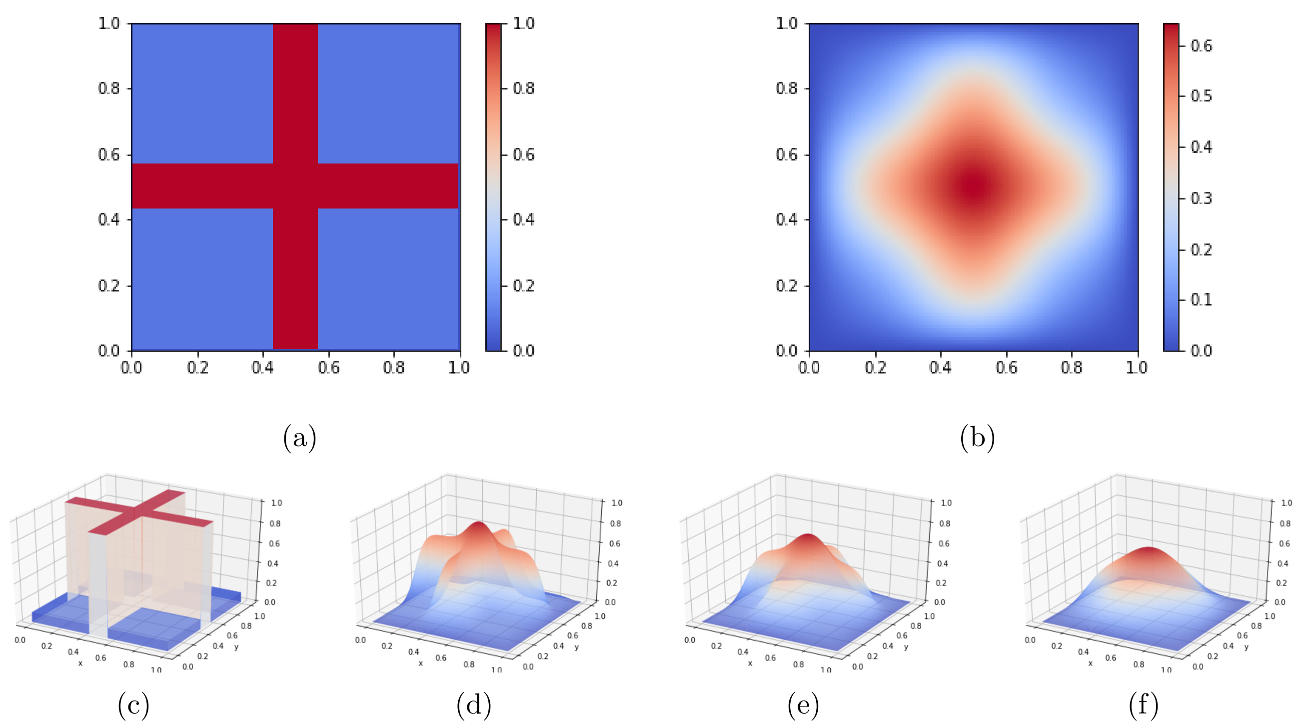


Figure 6: Cross diffusion. (a) Start at $t = 0$ ms. (b) Finish at $t = 4.71$ ms with 4000 iterations. (c-f) 3D visual of $u(x, y, t)$ at $t = 0$ ms, 2.35 ms, 4.71 ms, 9.41 ms and iteration count 0, 1000, 2000 and 4000 respectively.