Solving 2D Heat Equation Numerically using Python

G. Nervadof · Follow
Published in Level Up Coding
6 min read · Oct 13, 2020

When I was in college studying physics a few years ago, I remember there was a task to solve heat equation analytically for some simple problems. In the next semester we learned about numerical methods to solve some partial differential equations (PDEs) in general. It's really interesting to see how we could solve them numerically and visualize the solutions as a heat map, and it's really cool (pun intended). I also remember, in the previous semester we learned C programming language, so it was natural for us to solve PDEs numerically using C although some students were struggling with C and not with solving the PDE itself. If I had known how to code in Python back then, I would've used it instead of C (I am not saying C is bad though). Here, I am going to show how we can solve 2D heat equation numerically and see how easy it is to "translate" the equations into Python code.

Before we do the Python code, let's talk about the heat equation and finite-difference method. Heat equation is basically a partial differential equation, it is

$$\frac{\partial u}{\partial t} - \alpha \nabla u = 0$$

If we want to solve it in 2D (Cartesian), we can write the heat equation above like this

$$\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x} + \frac{\partial^2 u}{\partial y} \right) = 0$$

where u is the quantity that we want to know, t is for temporal variable, x and y are for spatial variables, and α is diffusivity constant. So basically we want to find the solution u everywhere in x and y, and over time t.

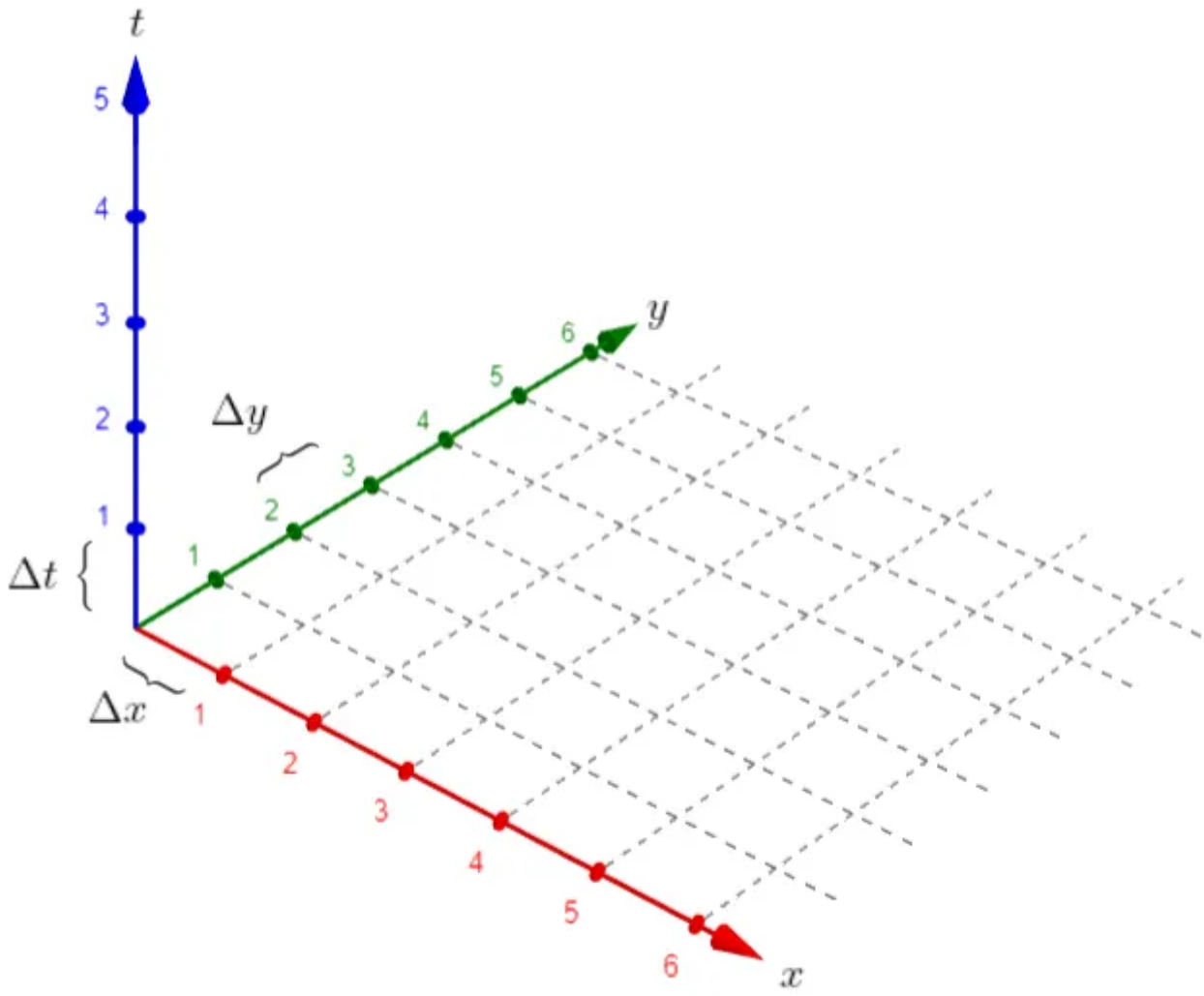
Now let's see the finite-difference method (FDM) in a nutshell. Finite-difference method is a numerical method for solving differential equations by approximating derivative with finite differences. Remember that the definition of derivative is

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

In finite-difference method, we approximate it and remove the limit. So, instead of using differential and limit symbol, we use delta symbol which is the finite difference. Note that this is oversimplified, because we have to use Taylor series expansion and derive it from there by assuming some terms to be sufficiently small, but we get the rough idea behind this method.

$$f'(a) \approx \frac{f(a+h) - f(a)}{h}$$

In finite-difference method, we are going to "discretize" the spatial domain and the time interval x, y, and t. We can write it like this



Cartesian coordinate, where x and y axis are for spatial variables, and t for temporal variable (coordinate axes from GeoGebra, edited by author)

$$x_i = i\Delta x$$

$$y_j = j\Delta y$$

$$t_k = k\Delta t$$

As we can see, i, j, and k are the steps for each difference for x, y, and t respectively. What we want is the solution u, which is

$$u(x, y, t) = u_{i,j}^{k}$$

Note that k is superscript to denote time step for u. We can write the heat equation above using finite-difference method like this

$$\frac{u_{i,j}^{k+1} - u_{i,j}^{k}}{\Delta t} - \alpha \left(\frac{u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}}{\Delta x^2} + \frac{u_{i,j+1}^{k} - 2u_{i,j}^{k} + u_{i,j-1}^{k}}{\Delta y^2} \right) = 0$$

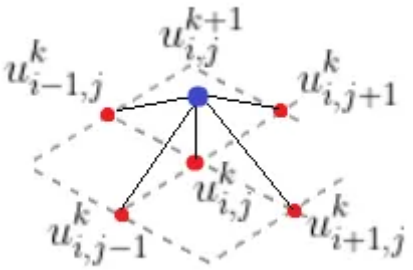
If we arrange the equation above by taking $\Delta x = \Delta y$, we get this final equation

$$u_{i,j}^{k+1} = \gamma(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k}) + u_{i,j}^{k}$$

where

$$\gamma = \alpha \frac{\Delta t}{\Delta x^2}$$

We can use this stencil to remember the equation above (look at subscripts *i*, *j* for spatial steps and superscript *k* for the time step)

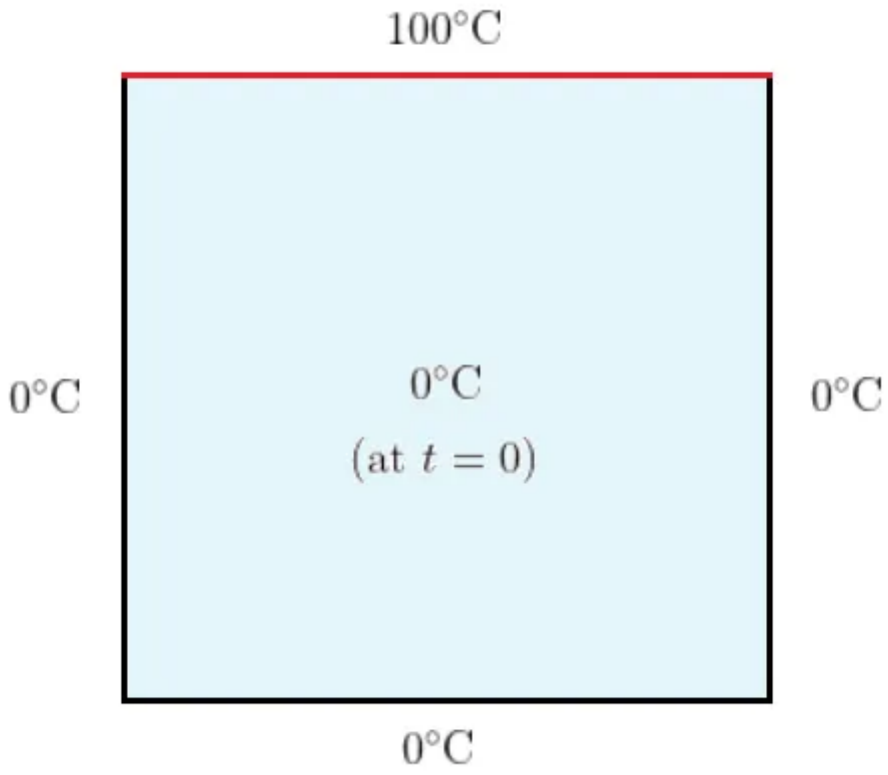


Explicit method stencil (image by author)

We use explicit method to get the solution for the heat equation, so it will be numerically stable whenever

$$\Delta t \leq \frac{\Delta x^2}{4\alpha}$$

Everything is ready. Now we can solve the original heat equation approximated by algebraic equation above, which is computer-friendly. For an exercise problem, let's suppose a thin square plate with the side of 50 unit length. The temperature everywhere inside the plate is originally 0 degree (at $t = 0$), let's see the diagram below (this is not realistic, but it's good for exercise)



Boundary and initial conditions for our exercise (image by author)

For our model, let's take $\Delta x = 1$ and $\alpha = 2.0$. Now we can use Python code to solve this problem numerically to see the temperature everywhere (denoted by i and j) and over time (denoted by k). Let's first import all of the necessary libraries, and then set up the boundary and initial conditions.

```python
# We use numpy (for array related operations) and matplotlib (for plotting)
# because they will help us a lot
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import FuncAnimation

plate_length = 50
max_iter_time = 1000

alpha = 2.0
delta_x = 1

# Calculated params
delta_t = (delta_x ** 2)/(4 * alpha)
gamma = (alpha * delta_t) / (delta_x ** 2)

# Initialize solution: the grid of u(k, i, j)
u = np.empty((max_iter_time, plate_length, plate_length))

# Initial condition everywhere inside the grid
u_initial = 0.0

# Boundary conditions (fixed temperature)
u_top = 100.0
u_left = 0.0
u_bottom = 0.0
u_right = 0.0

# Set the initial condition
u.fill(u_initial)

# Set the boundary conditions
u[:, (plate_length-1):, :] = u_top
u[:, :, :1] = u_left
u[:, :1, 1:] = u_bottom
u[:, :, (plate_length-1):] = u_right
```

fdm_2d_heat_equation_conditions.py hosted with ❤ by GitHub view raw

We've set up the initial and boundary conditions, let's write the calculation function based on finite-difference method that we've derived above.

```
def calculate(u):
  for k in range(0, max_iter_time-1, 1):
    for i in range(1, plate_length-1, delta_x):
      for j in range(1, plate_length-1, delta_x):
        u[k + 1, i, j] = gamma * (u[k][i+1][j] + u[k][i-1][j] + u[k][i][j+1] + u[k][i][j-1] - 4*

  return u
```

fdm_2d_heat_equation_algebraicalc.py hosted with ❤ by GitHub view raw

Let's prepare the plot function so we can visualize the solution (for each k) as a heat map. We use Matplotlib library, it's easy to use.

```
def plotheatmap(u_k, k):
  # Clear the current plot figure
  plt.clf()
  plt.title(f"Temperature at t = {k*delta_t:.3f} unit time")
  plt.xlabel("x")
  plt.ylabel("y")

  # This is to plot u_k (u at time-step k)
  plt.pcolormesh(u_k, cmap=plt.cm.jet, vmin=0, vmax=100)
  plt.colorbar()

  return plt
```

fdm_2d_heat_equation_plothm.py hosted with ❤ by GitHub view raw

One more thing that we need is to animate the result because we want to see the temperature points inside the plate change over time. So let's create the function to animate the solution.

```
def animate(k):
  plotheatmap(u[k], k)

anim = animation.FuncAnimation(plt.figure(), animate, interval=1, frames=max_iter_time, repeat=F
anim.save("heat_equation_solution.gif")
```

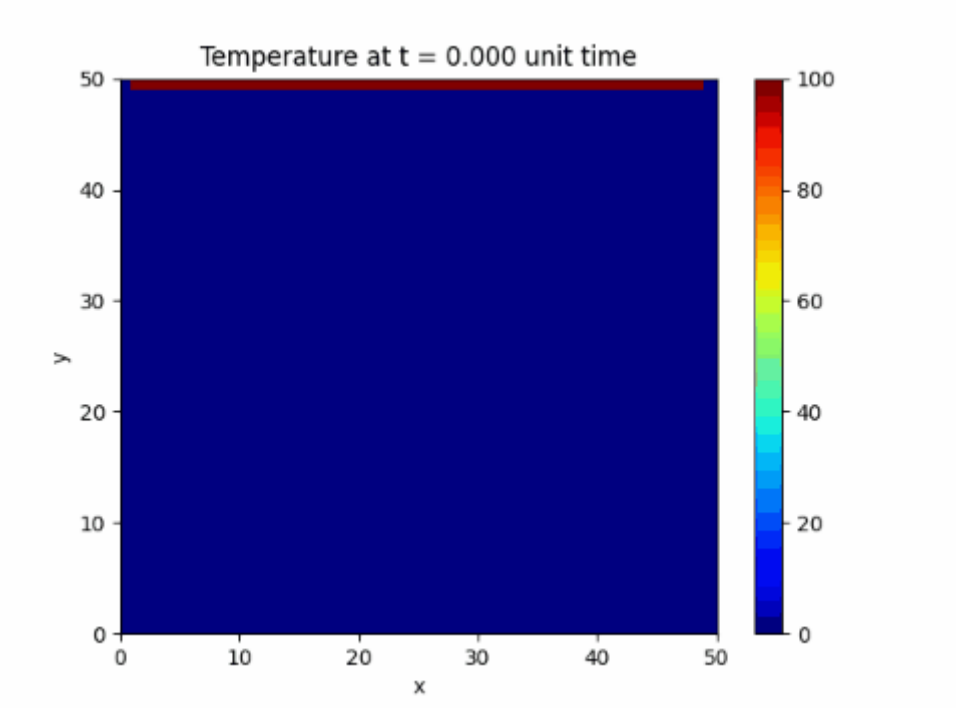
fdm_2d_heat_equation_animate.py hosted with ❤ by GitHub view raw

Now, we're done! Let's see the complete code below and run it.

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib.animation import FuncAnimation

print("2D heat equation solver")

plate_length = 50
max_iter_time = 750

alpha = 2
delta_x = 1

delta_t = (delta_x ** 2)/(4 * alpha)
gamma = (alpha * delta_t) / (delta_x ** 2)

# Initialize solution: the grid of u(k, i, j)
u = np.empty((max_iter_time, plate_length, plate_length))

# Initial condition everywhere inside the grid
u_initial = 0

# Boundary conditions
u_top = 100.0
u_left = 0.0
u_bottom = 0.0
u_right = 0.0

# Set the initial condition
u.fill(u_initial)

# Set the boundary conditions
u[:, (plate_length-1):, :] = u_top
u[:, :, :1] = u_left
u[:, :1, 1:] = u_bottom
u[:, :, (plate_length-1):] = u_right

def calculate(u):
    for k in range(0, max_iter_time-1, 1):
        for i in range(1, plate_length-1, delta_x):
            for j in range(1, plate_length-1, delta_x):
                u[k + 1, i, j] = gamma * (u[k][i+1][j] + u[k][i-1][j] + u[k][i][j+1] + u[k][i]|

    return u

def plotheatmap(u_k, k):
    # Clear the current plot figure
    plt.clf()
```

```
        plt.clf()

        plt.title(f"Temperature at t = {k*delta_t:.3f} unit time")
        plt.xlabel("x")
        plt.ylabel("y")

        # This is to plot u_k (u at time-step k)
        plt.pcolormesh(u_k, cmap=plt.cm.jet, vmin=0, vmax=100)
        plt.colorbar()

        return plt

    # Do the calculation here
    u = calculate(u)

    def animate(k):
        plotheatmap(u[k], k)

    anim = animation.FuncAnimation(plt.figure(), animate, interval=1, frames=max_iter_time, repeat=
    anim.save("heat_equation_solution.gif")

    print("Done!")
```

That's it! And here's the result



The numeric solution of our simple heat equation exercise

Cool isn't it? By the way, you can try the code above using this Python Online Compiler https://repl.it/languages/python3, make sure you change the *max_iter_time*

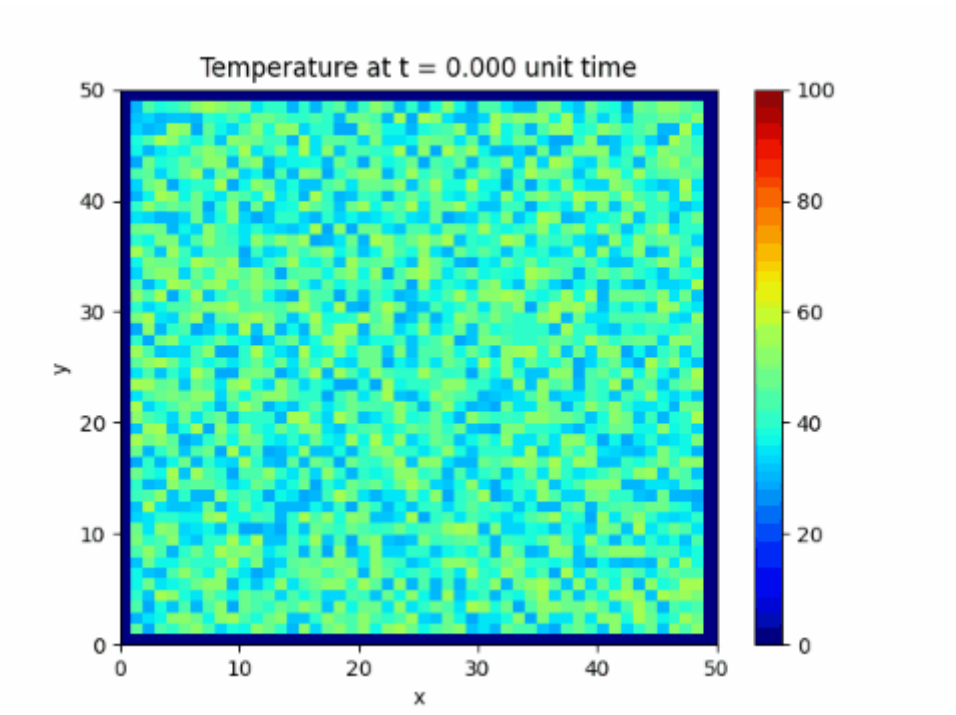
to 50 before you run the code to make the iteration result faster.

Okay, we have the code now, let's play with something more interesting, let's set all of the boundary conditions to 0, and then randomize the initial condition for the interior grid.

```
# Change boundary conditions
u_top = 0.0
u_left = 0.0
u_bottom = 0.0
u_right = 0.0

# Change u_initial (random temperature between 28.5 and 55.5 degree)
#u_initial = 0
u_initial = np.random.uniform(low=28.5, high=55.5, size=(plate_length,plate_length))

# Change initial conditions
#u.fill(u_initial)
u[0,:,:] = u_initial
```

fdm_2d_heat_equation_rand.py hosted with ❤ by GitHub view raw

And here's the result



The numeric solution where all boundary conditions are 0 with randomized initial condition inside the grid

Python is relatively easy to learn for beginners compared to other programming languages. I would recommend to use Python for solving computational problems

like we've done here, at least for prototyping because it has really powerful numerical and scientific libraries. The community is also growing bigger and bigger, and that can make things easier to Google when we're stuck with it. Python may not be as fast as C or C++, but using Python we can focus more on the problem solving itself rather than the language, of course we still need to know the Python syntax and its simple array manipulation to some degree, but once we get it, it will be really powerful.

. . .

Python | Data Visualization | Numerical Analysis | Physics

Follow

Written by G. Nervadof

151 Followers · Writer for Level Up Coding

Living in the emerald of the equator, believes that science and technology can and should make the world a better place | Unity in diversity