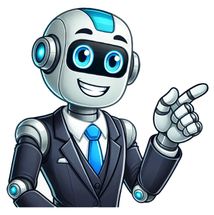
I'm not a bot



Finite difference method python heat equation

Differential equations are mathematical expressions that connect a function to its derivative, describing how one quantity changes in relation to another. They can be classified into two main categories: ordinary differential equations (ODEs), which depend on a single variable, and partial differential equations (PDEs), which involve multiple variables. Laplace's equation is an example of a PDE, while Maxwell's equations, including Gauss's Law, are also PDEs that describe fundamental physical phenomena such as electromagnetism. In physics, particularly in electromagnetism, we have derived an equation known as Laplace's equation. This equation represents the relationship between the electric potential φ and its second derivative. However, solving this partial differential equation analytically is extremely challenging. To overcome this difficulty, we can use approximations to simplify the problem. One approach is to use finite differences, which involve calculating the difference quotient of a function at two nearby points. We have derived two such finite differences for first-order derivatives: a forward and backward difference. By combining these finite differences, we can obtain a second-order central finite difference, which has a better approximation of the true derivative. To further improve accuracy, we can combine multiple finite differences in different directions to create higher-order approximations. The key idea here is to reduce the complexity of the original partial differential equation by discretizing it into smaller, more manageable components. By doing so, we can use numerical methods and approximations to find approximate solutions that are close enough to the exact solution for most purposes. In this case, we have applied these finite difference techniques to solve the given partial differential equation, which represents a fundamental problem in physics. The resulting equations will be useful for modeling and analyzing various physical phenomena, such as electromagnetic fields and potential distributions. The five-point stencil is used to approximate the solution of Laplace's equation. It involves setting up a grid with points at equal intervals, denoted by h. The stencil can be expressed as: \(f(x,y) \approx \frac{1}{4} (f(x+1, y)+f(x-1, y)+f(x, y+1)+f(x, y-1))\). This approach is equivalent to taking the average of the points in cardinal directions from the point in question. However, when dealing with a grid, it's essential to consider the boundaries and how they affect the solution. To address this issue, ghost points are introduced as fictitious points that pad the boundaries of the solution. These points are crucial for ensuring that the finite difference behaves correctly even at the edges of the domain. Two types of boundary conditions are commonly used: Dirichlet and Neumann. The Dirichlet condition sets the value of the boundary to a specific function, while the Neumann condition defines the derivative of the boundary. In the case where the derivative is zero, it's essential to determine the correct value for the ghost points. For a center point in a 2D solution, the expression can be rewritten as: \(f(x,y) \approx \frac{1}{4} (f(x+1, y)+f(x-1, y)+f(x, y+1)+f(x, y-1))\). By applying this formula and considering the boundary conditions, it's possible to determine any given point in a 2D solution to Laplace's equation. The authors are working with an image that has reached its upper boundary, where the value of y+h is out of bounds. To solve for this point, they can use the previously discussed method of substituting f(x, y) and solving for it using finite differences or convolution. Convolution is a mathematical operation that combines two matrices by sliding one matrix over the other and multiplying corresponding values together. In this context, the authors use convolution to approximate the Laplace operator (∇²) on an image. The kernel used in convolution is: (1/4) · ⎡ 0 1 0 ⎣ 1 0 1 ⎢ 0 1 0 ⎣ This kernel, when convolved with the initial conditions and the PDE, will satisfy Laplace's equation. However, this method only works for points that don't touch a boundary. To correct for edge factors, the authors use another matrix (the "Hadamard product") to adjust the scaling of the points on the boundary. This ensures that the resulting image aligns with the formal definitions of the Laplace operator. The process requires two passes: the first pass convolves the image with the kernel, and the second pass corrects for edge factors by taking the Hadamard product between the corrected image and the kernel. Given text: ``` 0 1/4 0 0 0 0 0 0 0 ``` Text paraphrased: The initial configuration of the system is provided, where certain values are set to specific fractions and zeros. This tutorial series on simulation in Python will focus on numerical simulations, with some exceptions like stable fluid and n-body solvers. The first step is understanding differential equations, which "nature speaks" in its language of math. While calculus isn't necessary for completing the tutorial, it's essential to grasp concepts like derivatives and integrals. For a beginner-friendly introduction, consider watching 3Blue1Brown's video on the heat equation. This partial differential equation (PDE) describes complex phenomena like heat diffusion and transfer. The solution, u(t, x), gives the temperature at any point along the rod x at time t. To solve the heat equation numerically, one must approximate the exact analytic solution. Various methods can be employed, including finite differences or finite elements. This tutorial will explore these approaches in detail, culminating in a demonstration of solving the 1D heat equation using Python code. ####The Heat Equation Consider a metal rod of length L with one end heated to 200°C and the other kept cold at 0°C. Over time, heat would spread across the rod, eventually reaching equilibrium. The heat equation describes this process, providing a solution for u(t, x) that gives the temperature at any point along the rod x at any given time t. Numerical methods will be used to approximate the solution of the heat equation, as the exact analytic solution is often difficult or impossible to obtain. This tutorial aims to provide a clear and concise explanation of these numerical methods, allowing readers to implement them in Python code. The Finite-Difference Method is an approach used to approximate the derivative of an unknown function f(x) by utilizing neighboring values. It allows us to estimate the tangent line to f(x) at a point n, which can then be used to calculate the function's derivative. By examining a point before n and after n, we can start estimating the tangent line. Using the neighboring points in sequence, such as n+1 and n-1, we refine our estimation until it closely resembles the actual tangent line. The central difference method provides the most accurate estimate by using both the forward and backward neighbors. Looking at how a derivative can be approximated using finite differences, it becomes clear that the formula works similarly to its algebraic definition. By taking the limit as dx approaches zero, this shows our answer is correct even further. When looking back at the heat equation and substituting the left-hand side with its finite difference version, we leave x untouched while differentiating with respect to time. The right-hand side of the equation then needs to be approximated using a finite difference formula. We can easily get a second-order approximation by applying a first-order one. Substituting this into our heat equation results in an interesting final form that we can solve for u(t+dt, x). From here, rearranging terms allows us to find the solution for u(t+dt, x), and with initial conditions, we can integrate our way through time, effectively solving any physics-based differential equation. 1. to always be temp_at_left_end and temp_at_right_end respectively. The initial conditions u(t, 0) = 200 and u(t, length) = 200 for all times t can be expressed in code as u[:, 0] = temp_at_left_end # u(t, 0) = 200u[:, -1] = temp_right_end # u(t, length) = 200 2. The plot of the initial u[0] shows high temperatures at both ends and zero everywhere else. 3. To solve the problem, we iterate through all points along the rod and in time, plugging them into the derived equation. ```python for t in range(1, len(t_vec)-1): for x in range(1, len(x_vec)-1): u[t+1, x] = k * (dt / dx**2) * (u[t, x+1] - 2*u[t, x] + u[t, x-1]) + u[t, x] ``` 4. Notice how the equation substitutes dx and dt with +1 or -1. 5. The numpy savvy among you would wonder why we're using nested for loops instead of vector notation, and the answer is simply for clarity. 6. To visualize the solution, call pyplot.plot() inside the loop for an animated plot or outside it for a static one. ```python pyplot.plot(u[t+1, x], label='Temperature at time t') ``` 7. The temperatures slowly equalize over time as heat spreads from both ends into the rod. 8. We can rewrite the time derivative using a backward difference, which makes the method implicit : $$ \frac{u_{i,j}^{k-1} - u_{i,j}^k}{\Delta t} \approx \frac{\partial u}{\partial t} $$ The full equation becomes: $$ \frac{u_{i,j}^{k-1} - u_{i,j}^k}{\Delta t} - \alpha \left(\frac{u_{i+1,j}^k - 2 u_{i,j}^k + u_{i-1,j}^k}{\Delta x^2} + \frac{u_{i,j+1}^k - 2 u_{i,j}^k + u_{i,j-1}^k}{\Delta y^2} \right) = 0 $$ 9. This equation is true for all samples i,j, so we can write it for all samples, and we get a system of N-samples equations that link all heat points of time k. 10. The system of equation is linear and can be solved using classic linear algebra to inverse the matrix of the system. The one-dimensional equation for three samples at time step k is given by: $$\frac{u_i^{k-1} - u_i^k}{\Delta t} - \alpha \left(\frac{u_{i+1}^k - 2 u_{i}^k + u_{i-1}^k}{\Delta x^2} \right) = 0$$ This can be simplified to: $$u_i^{k-1} - u_i^k = \alpha\frac{\Delta t}{\Delta x^2} \left(u_{i+1}^k - 2 u_{i}^k + u_{i-1}^k \right)$$ Simplifying further gives: $$u_i^k (2 \gamma - 1) = \gamma u_{i+1}^k + \gamma u_{i-1}^k - u_{i}^{k-1}$$ This equation can be applied to the central three samples with known boundary conditions at i=0 and i=4. The resulting system of equations is written in matrix form. The Crank-Nicolson method combines the explicit and implicit methods by approximating the time derivative using a central difference equation and solving for the unknown values at the half-time step. This involves splitting the spatial derivatives into two terms, one using the previous time step and one using the next time step. By propagating the system in this manner, the Crank-Nicolson method reduces to a linear system of equations that can be solved using matrix methods, similar to the implicit method.