Assignment 1

CASPER BRESDAHL, whs715, University of Copenhagen, Denmark

1 FINITE DIFFERENCE METHODS

Finite difference methods are methods for solving ordinary or partial differential equations by making use of Taylor's Theorem to approximate derivatives with finite differences. This is done by discretizing the domain into a finite number of steps and then the solution of these discrete points are approximated by solving equations of finite differences. The idea can be described as replacing the derivatives by finite difference approximations. For this assignment our focus is on partial differential equations, and we will thus not discuss ordinary differential equations further.
We can now imagine some function partial differential equation$f(x, y)$. If we discretize f we sample a finite number of values along both the x and y axis and we denote this spacing by respectively Δx and Δy. If we keep the spatial information of sampling f every Δx and Δy unit in mind, we can think of this discretization as constructing a computational grid, or just grid for short, where each node in the grid is a functional value. Increasing the sampling rate, i.e decreasing Δx and Δy, results in a larger and finer grid but also in a better approximation to f, whereas decreasing the sampling rate results in a smaller and coarser grid. The 'area' in which we sample is called the domain, and the partial differential equation, in our case f, which defines what happens in the domain is called a governing equation. With our grid we now have a notion of 'neighbours' and we can now look at how to approximate the derivatives of our grid by finite differences derived from Taylor's Theorem.

1.1 Forward, backward and central difference approximations

Assuming the function whose derivatives are to be approximated is properly behave we can construct a Taylor series:

$$f(x_i+\Delta x) = f(x_i)+\frac{f^{(1)}(x_i)}{1!}\Delta x+\frac{f^{(2)}(x_i)}{2!}\Delta x^2+\cdots+\frac{f^{(n)}(x_i)}{n!}\Delta x^n+\mathbf{o}(\Delta x)$$

It is this series we will use to construct finite difference approximations. If we wish to compute $\frac{\partial f}{\partial x}$ we use a first order Taylor expansion, that is, we use the above Taylor series, but truncate it after the first derivative:

$$f(x_i + \Delta x) = f(x_i) + \frac{f^{(1)}(x_i)}{1!}\Delta x + \mathbf{o}(\Delta x) \tag{1}$$

Using Equation 1 we can now solve for the first derivative:

$$\begin{aligned} f(x_i + \Delta x) &= f(x_i) + \frac{f^{(1)}(x_i)}{1!}\Delta x + \mathbf{o}(\Delta x) \iff \\ \frac{f(x_i + \Delta x)}{\Delta x} &= \frac{f(x_i)}{\Delta x} + f^{(1)}(x_i) + \frac{\mathbf{o}(\Delta x)}{\Delta x} \iff \\ f^{(1)}(x_i) &= \frac{f(x_i + \Delta x)}{\Delta x} - \frac{f(x_i)}{\Delta x} - \frac{\mathbf{o}(\Delta x)}{\Delta x} \end{aligned}$$

If we now disregard $-\frac{\mathbf{o}(\Delta x)}{\Delta x}$ we get an approximation known as the *forward difference approximation*:

$$\frac{\partial}{\partial x} f(x_i) \approx \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x}$$

Similarly we can write the first order Taylor expansion as:

$$f(x_i - \Delta x) = f(x_i) - \frac{f^{(1)}(x_i)}{1!}\Delta x + \mathbf{o}(\Delta x) \tag{2}$$

Using Equation 2 we can solve for the first derivative:

$$\begin{aligned} f(x_i - \Delta x) &= f(x_i) - \frac{f^{(1)}(x_i)}{1!}\Delta x + \mathbf{o}(\Delta x) \iff \\ \frac{f(x_i - \Delta x)}{\Delta x} &= \frac{f(x_i)}{\Delta x} - f^{(1)}(x_i) + \frac{\mathbf{o}(\Delta x)}{\Delta x} \iff \\ f^{(1)}(x_i) &= \frac{f(x_i)}{\Delta x} - \frac{f(x_i - \Delta x)}{\Delta x} + \frac{\mathbf{o}(\Delta x)}{\Delta x} \end{aligned}$$

If we now disregard $\frac{\mathbf{o}(\Delta x)}{\Delta x}$ we get an approximation known as the *backward difference approximation*:

$$\frac{\partial}{\partial x} f(x_i) \approx \frac{f(x_i) - f(x_i - \Delta x)}{\Delta x}$$

We can now also add the forward difference approximation and the backward difference approximation together:

$$2\frac{\partial}{\partial x} f(x_i) \approx \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} + \frac{f(x_i) - f(x_i - \Delta x)}{\Delta x} \iff \tag{3}$$

$$\frac{\partial}{\partial x} f(x_i) \approx \frac{1}{2}\frac{f(x_i + \Delta x) - f(x_i) + f(x_i) - f(x_i - \Delta x)}{\Delta x} \iff \tag{4}$$

$$\frac{\partial}{\partial x} f(x_i) \approx \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2\Delta x} \tag{5}$$

Equation 5 is known as the *central difference approximation.*

1.2 Higher order and higher dimensions approximations

We now turn our attention to higher order derivatives. If we want to compute $\frac{\partial^2 f}{\partial x^2}$ we can use the central difference approximation recursively, and use mid points to evaluate the finite differences:

$$\begin{aligned} \frac{\partial^2}{\partial x^2} f(x_i) &\approx \frac{\frac{\partial}{\partial x} f(x_i + \frac{1}{2}\Delta x) - \frac{\partial}{\partial x} f(x_i - \frac{1}{2}\Delta x)}{\Delta x} \\ &= \frac{\frac{f(x_i+\Delta x)-f(x_i)}{\Delta x} - \frac{f(x_i)-f(x_i-\Delta x)}{\Delta x}}{\Delta x} \\ &= \frac{f(x_i + \Delta x) - 2f(x_i) + f(x_i - \Delta x)}{\Delta x^2} \end{aligned}$$

For higher dimensions we get more or less the same expressions with the notation slightly changed. Taking the central difference approximation gives us:

$$\begin{aligned} \frac{\partial f(x_i, y_j)}{\partial x} &\approx \frac{f(x_i + \Delta x, y_j) - f(x_i - \Delta x, y_j)}{2\Delta x} \\ \frac{\partial f(x_i, y_j)}{\partial y} &\approx \frac{f(x_i, y_j + \Delta y) - f(x_i, y_j - \Delta y)}{2\Delta y} \end{aligned}$$

Author's address: Casper Bresdahl, whs715, University of Copenhagen, Copenhagen, Denmark, whs715@alumni.ku.dk.

We can also combine the higher order and higher dimensions to find:

$$\begin{aligned}\frac{\partial^2}{\partial y \partial x} f_{i,j} &= \frac{\partial}{\partial x}\left(\frac{\partial}{\partial y} f_{i,j}\right)\\ &\approx \frac{\partial}{\partial x}\left(\frac{f_{i,j+1} - f_{i,j-1}}{2\Delta y}\right)\\ &\approx \frac{f_{i+1,j+1} - f_{i-1,j+1} - f_{i+1,j-1} + f_{i-1,j-1}}{4\Delta x \Delta y}\end{aligned}$$

Where we here shorten the notation and have $f_{i,j} = f(x_i, y_j)$ and $f_{i+1,j+1} = f(x_i + \Delta x, y_j + \Delta y)$.

2 TOY EXAMPLE

We can now look at the toy example from the notebook and slides. We define $u(x, y) : \mathbb{R}^2 \to \mathbb{R}$ and we define the partial differential equation:

$$f(x, y) = \nabla^2 u - \kappa^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \kappa^2 u \tag{6}$$

Where f is known to us. As u is unknown we need a way to simplify the partial derivatives in Equation 6. This is where our finite difference approximations comes into the picture. Using (for instance) the central difference approximation we can now approximate the partial derivatives in terms of the surrounding nodes in our sampled grid. Using central difference approximation we can thus simplify Equation 6 to:

$$\begin{aligned}f_{i,j} &= \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} - \kappa^2 u\\ &= \frac{1}{\Delta x^2} u_{i-1,j} + \frac{1}{\Delta y^2} u_{i,j-1} + \left(-\kappa^2 - \frac{2}{\Delta x^2} - \frac{2}{\Delta y^2}\right) u_{i,j}\\ &\quad + \frac{1}{\Delta y^2} u_{i,j+1} + \frac{1}{\Delta x^2} u_{i+1,j}\end{aligned}$$

If we now substitute all the coefficients before the u terms with $c's$ (to shorten / generalize notation) we can derive *update formulas* for the u terms. We can for instance derive the update formula for the unknown $u_{i,j}$ as:

$$u_{i,j} = \frac{f_{i,k} - c_{i-1,j}u_{i-1,j} - c_{i,j-1}u_{i,j-1} - c_{i,j+1}u_{i,j+1} - c_{i+1,j}u_{i+1,j}}{c_{i,j}}$$

We here refer to f as the *source term*. From the update formula of $u_{i,j}$ we see which other u terms $u_{i,j}$ depends on. We can visualize this in terms of a *stencil* which more visually gives a notion of which other nodes in our grid our nodes depend on.

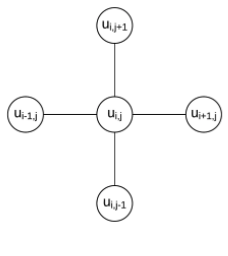


Fig. 1. Visualization of a stencil. Illustration taken from lecture slides.

In Figure 1 we can see an illustration of a stencil. We can imagine running this stencil over each node in our grid, which would then gives us the update formula for each node in the grid in terms of dependent nodes. However, we quickly realize that the stencil would stick outside our grid at the boundaries. We thus need some boundary conditions to handle what should happen at the boundaries.

2.1 Boundaries

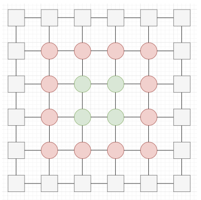


Fig. 2. Visualization of a grid with the green and red circles being domain nodes, red circles being boundary nodes and the square nodes being ghost nodes..

Two common boundary conditions are Dirichlet and von Neumann boundary conditions. We define them as:

$$\begin{aligned}u(x, y) &= k_d, \quad \forall x, y \in \Gamma\\ \frac{\partial u}{\partial x} &= k_n, \quad \forall x, y \in \Gamma\end{aligned}$$

Where Γ is the 'boundary of the boundary nodes' (the boundary nodes can be seen in Figure 2). We can apply one of two techniques to deal with boundary conditions, we either use elimination of unknown variables or we use ghost nodes. If we choose to apply a von Neumann boundary condition and we choose to apply elimination of variables to handle it, then for the left side of the grid, using central difference approximations, we will have:

$$\begin{aligned}\frac{\partial^2 u_{1,j}}{\partial x^2} &= \frac{\frac{\partial}{\partial x} u_{\frac{1}{2},j} - \frac{u_{2,j} - u_{1,j}}{\Delta x}}{\Delta x}\\ &= \frac{u_{1,j} - u_{2,j}}{\Delta x^2}\end{aligned}$$

As $\frac{\partial}{\partial x} u_{\frac{1}{2},j}$ is zero according to the von Neumann boundary condition. That is, we can compute the left boundary conditions as a negative forward difference approximation. Similarly, if $x_i = 5$ is our right boundary we have for the right boundary conditions:

$$\begin{aligned}\frac{\partial^2 u_{5,j}}{\partial x^2} &= \frac{\frac{u_{5,j} - u_{4,j}}{\Delta x} - \frac{\partial}{\partial x} u_{5\frac{1}{2},j}}{\Delta x}\\ &= \frac{u_{5,j} - u_{4,j}}{\Delta x^2}\end{aligned}$$

Which is a backwards difference approximation. This also extends symmetrically to the top and bottom boundary conditions.
If we choose to use the ghost node approach we then extends our grid. This can be seen in Figure 2 where the circles is our domain and the squares are 'extra' nodes which we append to ensure the stencil never goes out of bounds. To find the values of these ghost

nodes we can now 'unpack' the von Neumann boundary conditions from above:

$$\frac{\partial}{\partial x} u_{\frac{1}{2},j} = \frac{u_{1,j} - u_{o,j}}{\Delta x} = 0$$

This implies $u_{0,j} = u_{1,j}$ (we would obtain similar results if we looked at the other boundaries), that is our boundaries should 'reflect' or mirror the domain values it is 'facing'.

2.2 Solutions

Now that we have dealt with our boundaries, we can compute solutions in two different ways. First we can use our stencil, and iteratively go over all domain nodes in our grid updating each node according to our update formula. This requires we make an initial guess for the values of our nodes which simply could be all of them are equal to zero at iteration zero. We would keep updating our nodes until they 'converge', i.e. the nodes no longer change.

Another way to compute a solution is to note the stencils form linear systems of equations where we have:

$$f_{1,1} = c_{1,0}u_{1,0} + c_{0,1}u_{0,1} + c_{1,1}u_{1,1} + c_{2,1}u_{2,1} + c_{1,2}u_{1,2}$$

$$\vdots$$

$$\begin{aligned} f_{n-1,m-1} = {} & c_{n-1,m-2}u_{n-1,m-2} + c_{n-2,m-1}u_{n-2,m-1} \\ & + c_{n-1,m-1}u_{n-1,m-1} + c_{n,m-1}u_{n,m-1} + c_{n-1,m}u_{n-1,m} \end{aligned}$$

Where n and m are the size of the grid. We can similarly write a linear system for the boundary conditions. If we split the coefficients and the u values we can construct the matrix system $\mathbf{Au} = \mathbf{f}$ where $\mathbf{A}$ encodes the coefficients and $\mathbf{f}$ is our vectorized grid values. We can compute the vector index from the 2d grid indices as $k = jN + i$ where N is the number of columns in the grid. To construct $\mathbf{A}$ we similarly use the encoding $\mathbf{A}_{(jN+i),(bN+a)} = c_{a,b}$ where we would define a and b in relation to i and j, i.e. if we want to encode the middle node of our stencil, this would have index $\mathbf{A}_{jN+i),jN+i)}$ and the left arm of the stencil would have index $\mathbf{A}_{jN+i),((j-1)N+i)}$. This maps our stencils to rows of $\mathbf{A}$ as seen in Figure 3. After the encoding we can simply solve $\mathbf{Au} = \mathbf{f}$ for $\mathbf{u}$. This process is called *matrix assembly*.

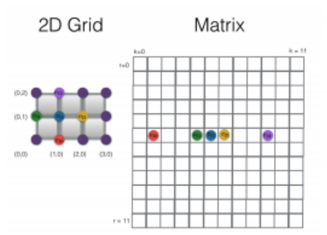
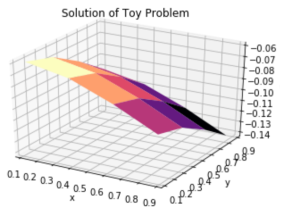


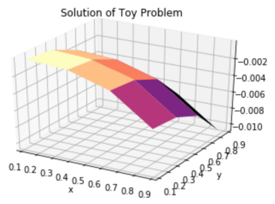
Fig. 3. Visualization of a grid with the green and red circles being domain nodes, red circles being boundary nodes and the square nodes being ghost nodes.

3 EXPERIMENTS

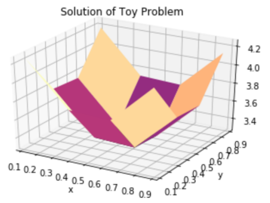
As a first experiment we can experiment with what happens when we change the values of f, i.e. our grid values, and when we change κ in our toy example. In the notebook we define $f = X + Y$ where X and Y are linear sequences of numbers. This defines a flat grid with a 'tilt'. If we now define $f = X^4 + Y^4$ the grid is no longer linear but instead gets curved as illustrated in Figure 4a. If we increase the value of κ we make the denominator in the update formula for $u_{i,j}$ smaller which results in $u_{i,j}$ increasing. If the grid is linear, we see this change simply 'shifts' the grid up, Figure 4d, whereas if the grid is non-linear we get more curve to the solution and the shift, Figure 4b. We can also change the von Neumann boundary condition such that the constant becomes 1. The results of this is the boundary of the solution becoming very steep as seen in Figure 4c.



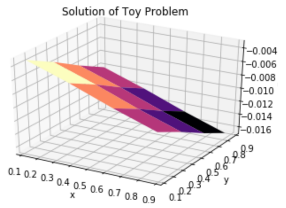
(a) Solution when the grid $f = X^4 + Y^4$.



(b) Solution when the grid $f = X^4 + Y^4$ and $\kappa = 10$.



(c) Solution when the grid $f = X + Y$, $\kappa = 2$ and von Neumann conditions equal 1.



(d) Solution when the grid $f = X + Y$ and $\kappa = 10$.

As another experiment, we can look at the time it takes to solve $\mathbf{Au} = \mathbf{f}$ for bigger and bigger square grids. We can do this by doing the matrix assembly and then time how long it takes numpy to perform *np.linalg.solve*. We do this over 10 iteration where we solve the same system and then take the average of the times to avoid some of the variance of how many resources the computer has available at the time of the computation. In Figure 5 we see that the time to solve the linear system grows exponentially as we increase the grid size. This is not entirely surprising as the number of grid cells in a square grid also grows exponentially, and so as we increase the grid size the $\mathbf{Au}$ matrix grows exponentially in size.

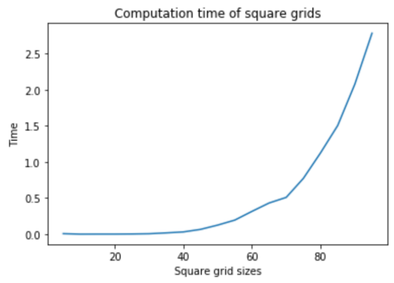


Fig. 5. Graph of how long it takes to solve the linear system vs. size of a square grid.