

# CMIS Hand-in 1: Finite Difference Methods 1

Nikolai Plambeck Nielsen

lpk331@alumni.ku.dk

Niels Bohr Institute, University of Copenhagen

## 1 INTRODUCTION

When solving differential equations on a computer, one has several options. In this hand-in the finite difference method (FDM) is explored.

In general we are interested in calculating the solution to some system, over some domain. This system is governed by a partial differential equation, and the domain is the “space” over which we solve the system, for example a line, surface or volume. We then sample this domain at a finite set of points, called the nodes of the system. The computational mesh is then the collection of all nodes in our domain.

For this hand in we are dealing with a regular mesh. This means that we sample the system at regular intervals in space, with a spacing of  $\Delta x$ . An illustration of this can be seen in figure 1.

The  $x$ -values are given by:

$$x_i = x' + i\Delta x \quad (1)$$

where  $x'$  is usually taken as the origin, for convenience, and where  $i \in \{0, 1, \dots, N\}$ . The sampled values of the system  $f(x_i)$  are then denoted in shorthand by

$$f_i \equiv f(x_i) \quad (2)$$

This of course generalises to higher dimensional systems. For two dimensions we have  $y_j = y' + j\Delta y$ , and  $f_{i,j} \equiv f(x_i, y_j)$ , and more indices for even higher dimensions.

## 2 THE FINITE DIFFERENCE METHOD

The idea behind the finite difference method is to replace all derivatives in the equations with differences instead. This can be seen as not letting the spacing  $h$  in the differential quotient go all the way to zero, but just to some small value to get a reasonable approximation of it. Since we are working with regular grids, a convenient value is the spacing between points  $\Delta x$ :

$$\frac{df(x_i)}{dx} = \lim_{h \rightarrow 0} \frac{f(x_i + h) - f(x_i)}{(x_i + h) - x_i} \rightarrow \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} \quad (3)$$

In this particular case, the finite difference is called the “forward difference” (FD) since it uses the point of interest  $x_i$  and the one next to it  $x_{i+1} = x_i + \Delta x$ . In the compact notation we have

$$\frac{df_i}{dx} \approx \frac{f_{i+1} - f_i}{\Delta x} \quad (4)$$

There is also the “backwards difference” (BD):

$$\frac{df_i}{dx} \approx \frac{f_i - f_{i-1}}{\Delta x} \quad (5)$$

And the “central difference” (CD):

$$\frac{df_i}{dx} \approx \frac{f_{i+1} - f_{i-1}}{2\Delta x} \quad (6)$$

These can be shown explicitly from the Taylor Polynomials for  $f(x)$  around  $x_i$ . There are of course also higher order differences. In

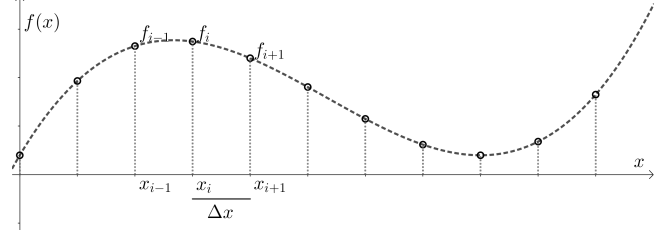


Figure 1: A regular grid of nodes for some function  $f$

particular we use the second order central difference when studying the heat equation, as this includes a second derivative:

$$\frac{d^2 f_i}{dx^2} \approx \frac{f_{i-1} - 2f_i + f_{i+1}}{\Delta x^2} \quad (7)$$

## 2.1 Restrictions on parameters

There are some considerations we need to make when designing the simulation of the system. In general we have some restrictions in the form of time, memory and accuracy.

These restrictions boil down to choosing the parameters of the system (for example  $\Delta x$  and  $\Delta y$  so as to get an acceptable trade-off between the restrictions).

The first concern is accuracy. We set out to solve some partial differential equation. This is all for naught, if we do not calculate an accurate result. The easiest way of increasing the accuracy is to decrease the spacing between the nodes. This of course results in more accurate approximations of the derivatives (with analytical results as  $\Delta x \rightarrow 0$ ), but also increases both computation time and memory cost (if we keep the domain constant).

One might then think, given infinite time and memory - could we just keep decreasing the grid spacing  $\Delta x$  to get a more accurate result? It turns out we cannot, as the computer cannot resolve the difference between values accurately, as they approach each other.

To be more specific, the computer stores the numbers as floating point numbers, which in Numpy has a spacing of  $2.2204 \cdot 10^{-16}$  (called the machine epsilon) for double precision floats (accessed by `numpy.finfo(float).eps`). Two numbers which differ by less than this value will be identical to the computer, and their difference will thus be 0. Trying to then divide by this difference (for example) will lead to crashing of the simulation.

## 3 CONSTRUCTING THE SIMULATION

To construct a simulation, one discretises the governing equation, after which one has a couple of options. Either one can isolate the unknown  $u_{i,j}$ , and loop over the computational mesh, updating each node in turn. Otherwise one can perform a matrix assembly of the system, where the discretised equation is expressed as a matrix equation to be solved, for example with a direct or iterative solver.

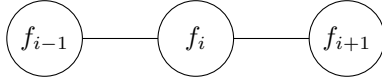


Figure 2: Stencil for the second order centred difference

In the discretisation step it can be useful to have a high-level overview of which values are needed to update each node. This is where the stencil comes in. This is a schematic representation of which nodes are involved in each calculation. If, for example we want to calculate the second order derivative of a function at the point  $x_i$ , using the second order centred difference, we will need the nodes  $x_i, x_{i+i}$  and  $x_{i-1}$ . This can be graphically shown as in figure 2, where each node is represented by a circle. Usually we denote the unknown in the system by  $u$  and reserve  $f$  for known functions, called the source terms.

#### 4 DEALING WITH BOUNDARY CONDITIONS

When on the boundary of the domain, ones options in discretisation is more limited. One cannot use the central difference, as there may be no node on one or the other side. Here we are presented with two major options: elimination of variables and using ghost notes.

Elimination of variables makes use of forward/backwards differences at the borders, such that only existing nodes are used, while using ghost nodes means appending additional nodes to the borders of the domain, such that central differences can be used throughout the computational mesh.

#### 5 A TOY PROBLEM

In this toy problem we apply the concepts described above. We start with the 2D governing equation

$$(\nabla^2 - \kappa^2)u(x, y) = f(x, y), \quad (8)$$

where  $\kappa > 0$  and  $f$  is a known source term. Our domain is the unit square, with 4 samples in each direction, so  $\Delta x = \Delta y = 1/3$ .

We apply the boundary conditions  $\partial u / \partial x = 0$  on the vertical boundaries, and  $\partial u / \partial y$  on the horizontal boundaries. We further let  $f(x, y) = x + y$  and  $\kappa = 2$ . We handle the boundary conditions with ghost notes.

As such our computational mesh consists of 36 nodes, as shown in figure 3, with the inner 16 nodes being the domain. In Python we start at the top left and then go down when looping over values of an array.

The only term in the governing equation that needs discretisation is the laplacian, which we approximate as a sum of two central second order differences:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad (9)$$

$$\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2}. \quad (10)$$

Collecting all the different terms we get

$$\sum_{k=-1}^1 \sum_{k'=-1}^1 c_{i+k,j+k'} u_{i+k,j+k'} = f_{i,j} \quad (11)$$

		i					
		0	1	2	3	4	5
j	0	0	1	2	3	4	5
	1	6	7	8	9	10	11
	2	12	13	14	15	16	17
	3	18	19	20	21	22	23
	4	24	25	26	27	28	29
	5	30	31	32	33	34	35

Figure 3: A graphic representation of the computational mesh.

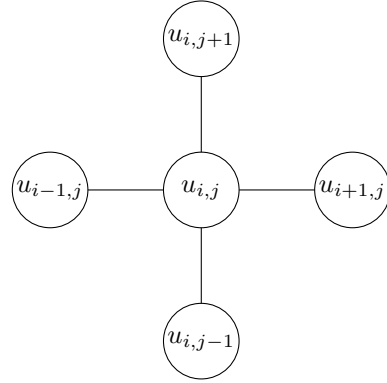


Figure 4: Stencil for the toy problem

with

$$c_{i-1,j} = c_{i+1,j} = \Delta x^{-2} \quad (12)$$

$$c_{i,j-1} = c_{i,j+1} = \Delta y^{-2} \quad (13)$$

$$c_{i,j} = -\kappa^2 - 2\Delta x^{-2} - 2\Delta y^{-2} \quad (14)$$

for a stencil as seen in figure 4

##### 5.1 Updating formulas

From here it is easy to isolate  $u_{i,j}$  to get the update formula for the domain nodes:

$$u_{i,j} = \frac{1}{c_{i,j}} \left( f_{i,j} - \sum_{k=-1,1} \sum_{k'=-1,1} c_{i+k,j+k'} u_{i+k,j+k'} \right) \quad (15)$$

For the boundary condition we need the first derivative on the boundary to be zero. Using a central difference on the left border we get

$$\frac{\partial u_{1,j}}{\partial x} = \frac{u_{2,j} - u_{0,j}}{2\Delta x} = 0 \Rightarrow u_{0,j} = u_{2,j}, \quad (16)$$

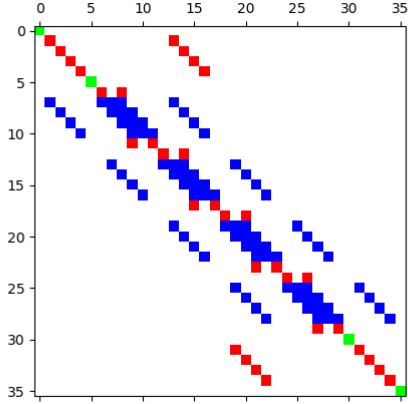


Figure 5: Spying on the matrix  $A$ . Rows with green correspond to corner ghost nodes. Red to regular ghost nodes and blue to nodes in the domain.

and similarly for the other boundaries, giving us the updating formula for the boundary nodes. To update the computational mesh we first update the ghost nodes, and after that the domain nodes using the above updating formulas.

## 5.2 Matrix Assembly

For the matrix assembly we stack up all the nodes in the 2D computational mesh  $u$  into a single 1D column vector, with the order given in figure 3. This is exactly the way `np.flatten` works. If we label the vector by  $n$ , then  $n = N_y i + j$ , where  $N_y$  is the number of nodes in the  $y$  direction. The coefficients for each node can then be written up in a row of an  $N_x N_y \times N_x N_y$  matrix  $A$ . The source term  $f$  is of course flattened in a similar way to the vector  $u$ .

Take the node  $i = 2, j = 3$ , as an example. This node is in the domain, with  $n = 6 \cdot 2 + 3 = 15$ . This then corresponds to the 16th row of  $A$ . Looking at the stencil we see that 5 entries of the row are populated,  $m \in \{9, 14, 15, 16, 21\}$ .

For the ghost nodes to follow the boundary conditions we need  $c_{0,j} = -c_{2,j}$  and  $f_{0,j} = 0$  for the matrix equation to hold (or similarly for the top/right/bottom border). This then ensures that the derivative at  $i = 1, j$  is zero (in the central difference approximation).

With these two specifications, 32 rows of the matrix have been specified. However, the rows corresponding to the corner ghost nodes have yet to be filled in. If they are not, then the matrix will be singular due to these rows of zeros and a direct solver like `np.linalg.solve` cannot be used. Since no other nodes use the values of the corner ghost nodes, we just set  $c_{i,j} = 1$  for each of the corner nodes. Spying on this matrix we see the following (figure 5) And indeed, we have four rows of domain nodes, each with 5 elements filled in, separated by two side-ghost nodes (which can be seen since there is an empty element between the 2 filled in). At the “ends” we have a corner, 4 ghost nodes and another corner.

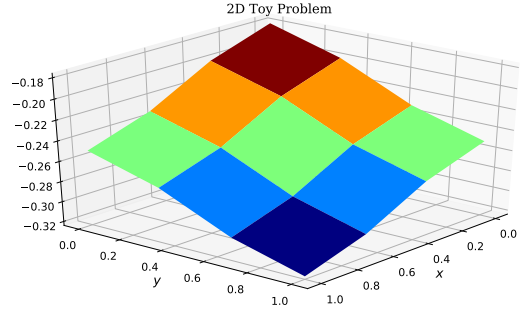


Figure 6: result from the toy problem, with  $n = 4$

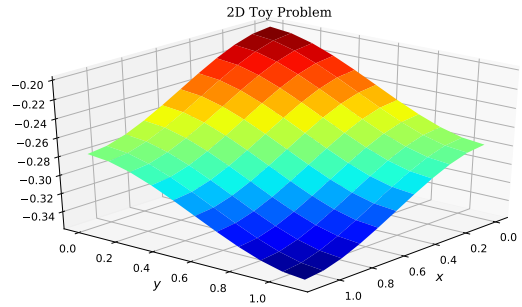


Figure 7: result from the toy problem, with  $n = 10$

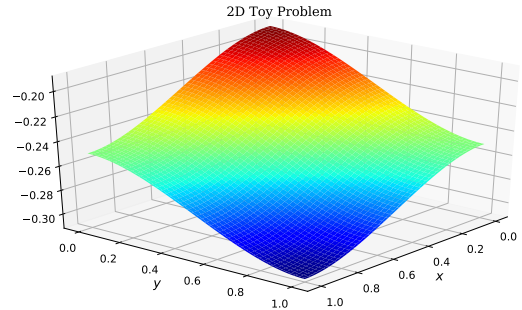


Figure 8: result from the toy problem, with  $n = 100$

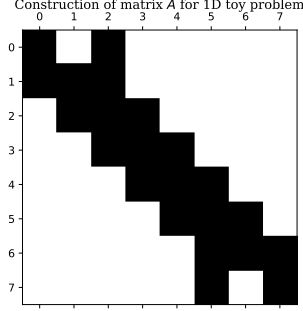
With the matrix assembled the system  $Au = f$  can be solved to get the solution  $u$ .

## 5.3 Results

After the equation has been solved, the result  $u$  has to be reassembled into a 2D array for plotting. This is done with the `np.reshape` command. With  $u$  back in the proper shape, we can plot the result (without the ghost nodes, of course). These are shown in figures 6, 7 and 8 with  $n = 4, 10, 100$  respectively.

$n$	min	max
4	-0.321	-0.179
10	-0.356	-0.200
100	-0.310	-0.190
150	-0.312	-0.191

**Table 1: Table of minimum and maximum values of  $u$**



**Figure 9: Structure of the matrix  $A$  for the 1D toy problem.**  
 $n = 6$

Our intuition says that the result should not depend highly on  $n$  (except in the pathological cases of  $n = 1$  where our approximations break down, and  $n \rightarrow \infty$  where numerical blow up will happen). To make this clear, the maximum and minimum value of  $u$  are tabulated in table 1 (the max and min happen at (0,0) and (1,1) respectively): This measure of the accuracy of the simulation might not be the best, but it is at least in the same order of magnitude for all  $n$  calculated.

## 6 A SECOND TOY PROBLEM

In the second toy problem we look at the governing equation

$$\frac{\partial^2 u}{\partial x^2} = 0 \quad (17)$$

on the domain  $x \in [0, 1]$ , with boundary conditions  $du/dx = 0$  at both edges, realised by ghost nodes, and  $n > 3$ .

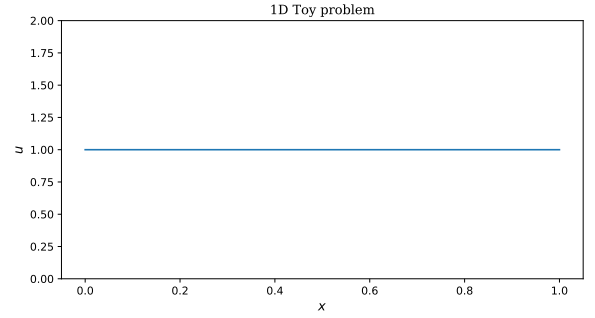
For this problem we again use a second order central difference for the discretisation, resulting in the stencil seen in figure 2. The matrix assembly is considerably easier as  $u$  is already a column vector.

For the boundary condition we use the central difference giving  $c_0 = c_{n+1} = (2\Delta x)^{-1}$  and  $c_2 = c_{n-1} = -(2\Delta x)^{-1}$ , with  $f_0 = f_{n+1} = 0$ . Spying on the matrix  $A$  gives the picture seen in figure 9. This matrix has rank 7 (and size 8), with eigenvalues tabulated in the first column of table 2. As seen, the seventh eigenvalue is of the order  $\epsilon$ , and is therefore most likely 0. This is to be expected for a matrix with rank  $n-1$ . The clue to construct a matrix for the system with full rank lies in analysing the governing equation. Integration twice gives

$$\iint \frac{\partial^2 u}{\partial x^2} dx = u = \frac{\partial u}{\partial x} x + a \quad (18)$$

2 derivative BCs	1 derivative BC
-94.917	-94.998
-80.825	-81.055
-60.863	-61.104
-39.296	-39.258
-20.394	-19.854
-7.379	-6.453
$1.938 \cdot 10^{-16}$	-0.616
-1.326	0.838

**Table 2: Table of eigenvalues for the matrix  $A$  with different boundary conditions**



**Figure 10: The 1D toy example, with a full rank matrix  $A$ .**

which is just a straight line. To fully specify the solution we need the derivative of the line, and some fixed point (like the intersection with the  $u$ -axis). Our boundary conditions gives us the derivative, but an intersection is still missing. Incorporating a different boundary condition on one side (say  $u_b = b$ ) would provide a fixed point, the specific solution, a full rank matrix and 8 non-zero eigenvalues.

As an example we set  $u_b = 1$  for the ghost node, corresponding to  $c_n = 1$  and  $f_{n+1} = 1$  for the ghost node row. The eigenvalues are tabulated in the second column of table 2. The result of this simulation is seen in figure 10, and sure enough  $u = 1$  for all  $x$ .