Homework 3

Arturo Popoli

November 16, 2025

The goal of this exercise is to build your own 2D FDM solver for Poisson-type equations on cartesian grids.

1 Meshgrid is your friend

- Define two arrays x and y to store the x and $y-$ coordinates of your grid nodes. Create and visualize a 5x7 nodes grid with $L_x = L_y = 1$ using the `meshgrid` function.

```
[X,Y] = meshgrid(x,y)
```

- Now you can plot (and check) your grid using

```
plot(X,Y,'b.')
```

  You should get something similar Fig. 1 (where I have added the node coordinate using the `text` function in a for loop).

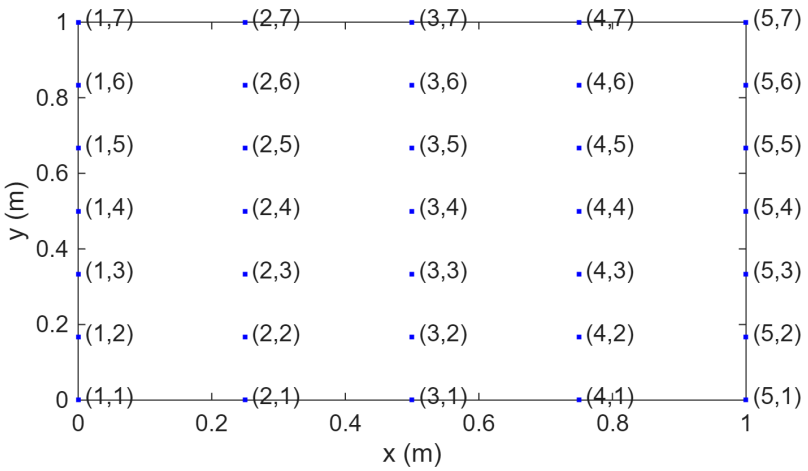


Figure 1: Our first 2D grid

2 Your first 2D FDM solver

The goal of this step is to create a function to solve Poisson-type problems on a 2D grid with zero Dirichlet BCs on every side of the domain and a uniform (i.e., constant) source term $t(x, y) = t$. This is the formulation of the problem:

$$\begin{cases} \dfrac{\partial^2 \varphi}{\partial x^2} + \dfrac{\partial^2 \varphi}{\partial y^2} - t = 0, & \\ \varphi(x, 0) = \varphi_S = 0 & \text{South edge}, \\ \varphi(0, y) = \varphi_W = 0 & \text{West edge}, \\ \varphi(L_x, y) = \varphi_E = 0 & \text{East edge}, \\ \varphi(x, L_y) = \varphi_N = 0 & \text{North edge}. \end{cases} \tag{1}$$

Create an m-function `FDM 2D s1` with the following I/O scheme:

- input: a `geom` structure containing info on the geometry and the discretization (L_x, L_y, number of nodes in the x and y-directions); a spatially-uniform source term `t`
- output: X,Y ("meshgridded" nodal coordinates), φ (computed values of unknown function)

Run the code with the following settings:

```
geom.nx = 25;
geom.Lx = 1;

geom.ny = 25;
geom.Ly = 1;
t = -1;

[phi,X,Y] = FDM_2D_s1(geom,t);
```

Use the `surf` function to plot the results

```
surf(X,Y,phi)
```

You should obtain something like this:

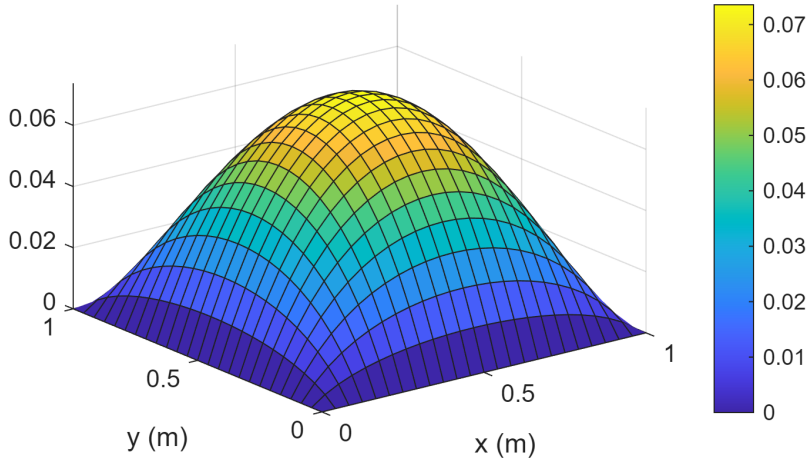


Figure 2: `FDM s1` - solution

Suggestion: You can use the following code structure.

```
function [phi,X,Y] = FDM_2D_s1(geom,t)
    %Remember to explain your functions for others but mainly for yourself in the future!
    %   geom: ...
    %   t: ...
    %   phi = ...
    %   X,Y = ...
    %   example of usage: ...

    % extract the info you need from geom
    ...

    % use meshgrid to create the X and Y matrices
    ...

    % preallocate [K] and {rhs} for speed using the zeros function
    ...
```

```
    % pre-define the coefficients for the [K] matrix
    kx = ... % horizontal coefficient
    ky = ... % vertical coefficient
    kc = ... % central coefficient

    % loop for [K] and {rhs} assembly
    for j = 1:ny % number of nodes along the y-direction
        for i = 1:nx % number of nodes along the x-direction
            % compute linear node index
            k = ...

            if i==1||i==nx||j==1||j==ny % boundary nodes
                K(k,k) = ...
                rhs(k) = 0; % boundary condition
            else
                % internal nodes
                K(k,k) = ...
                K(k,k+1) = ...
                K(k,k-1) = ...
                K(k,k+nx) = ...
                K(k,k-nx) = ...

                rhs(k) = ...
            end
        end
    end

    % solve linear system using \ (backslash) function
    phi = ...

    % the obtained phi array will have to be "reshaped" to get an nx*ny matrix
    % lookup the "reshape" function in the MATLAB documentation

end
```

2.1 Verification

Now we aim to verify the output of our code by comparing the numerical solution to some kind of analytical expressions.
Luckily, by applying the method of manufactured solutions one can show that – for a unit square domain with zero Dirichlet BCs – there is an exact solution when $t(x, y) = -8\pi^2 \sin(2\pi x) \sin(2\pi y)$, and this solution is:

$$\varphi_{\text{analytical}} = \sin(2\pi x) \sin(2\pi y), \tag{2}$$

which you can define as a function handle

```
phi_exact = @(x,y) sin(2*pi*x).*sin(2*pi*y).
```

Problem: so far the code accepts only uniform right-hand sides.

- Modify the `FDM_2D_s1` function to accept right-hand-sides defined via an anonymous function, e.g.,:

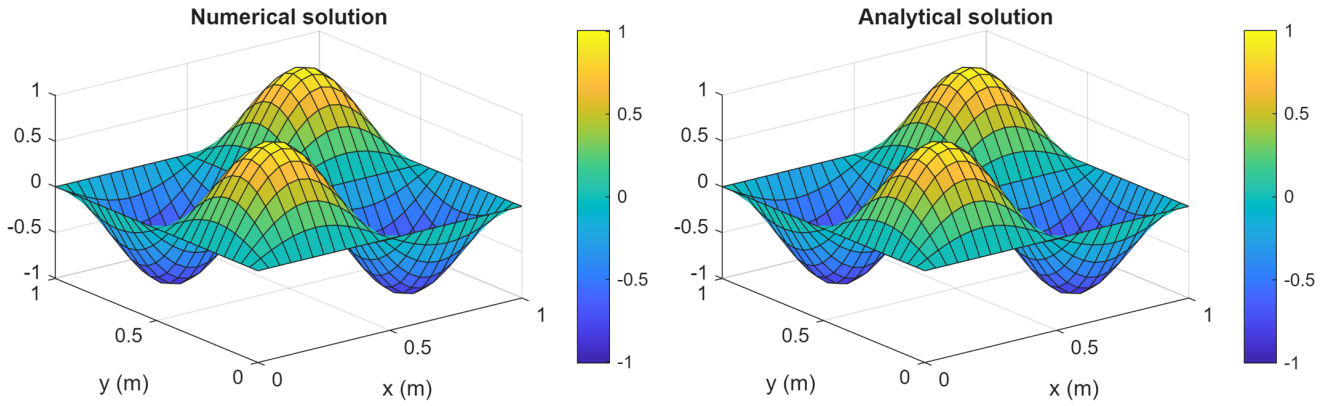
  ```
  t = @(x,y) -8*pi^2*sin(2*pi*x)*sin(2*pi*y)
  ```

Check that the numerical solution is coherent with the analytical one, using the following settings:

```
geom.nx = 25;
geom.Lx = 1;
```

```
geom.ny = 25;
geom.Ly = 1;
```

By plotting the numerical and the analytical solution you should get the following figures:



Now we want to check that the accuracy of our code improves when we increase the number of nodes.

- Plot the *error* given by the difference between the numerical and analytical solution

```
err = phi-phi_exact(X,Y);
```

- Check if (as expected) the error decreases when we increase `geom.nx` and `geom.ny`.

2.2 Back compatibility

Originally, our solver was designed to accept only constant (uniform) values of the source term t. We have now extended it to support spatially varying right-hand sides defined through function handles. However, for backward compatibility, we still want the solver to accept constant values as well.
Ideally, we would like the solver to handle **both** situations:

- if the user provides a constant value (e.g. `t = 3`), the code should treat it as a uniform source term;
- if the user provides a function handle (e.g. `t = @(x,y) ...`), the solver should evaluate it at the local grid coordinates.

To achieve this flexibility, we can add a small input-normalization step inside `FDM_2D_s1`. Check whether `t` is already a function handle; if not, wrap it into one:

```
% Ensure t is a function handle
if ~isa(t, "function_handle")
    t_value = t;           % copy the constant
    t = @(x,y) t_value;    % wrap it into a constant function
end
```

Check if our trick is working by running the `t = -1` example in 2.

3 Improving boundary conditions

Now we want to extend the capabilities of our code by allowing more flexible boundary conditions. Instead of fixing a single BC type for all boundaries, we want the user to specify, independently for each side of the domain, which boundary condition applies and what numerical value is imposed. To do that, store the boundary conditions in a structured variable `BC`, where each field corresponds to one side of the rectangular domain:

- `BCs.S` – South boundary

- `BCs.W` – West boundary
- `BCs.N` – North boundary
- `BCs.E` – East boundary

Each of these fields is itself a small structure containing two pieces of information:

- The boundary condition *type* (e.g. 'D' for Dirichlet or 'N' for Neumann)
- The boundary condition *value*, i.e. the constant value that should be imposed for that BC

This choice of organization has several advantages:

- It makes the solver easier to extend later. Indeed, additional BC types (Robin conditions, spatially varying data...) can be added by simply updating the structure without modifying the overall interface.
- It allows the user to specify different kinds of BCs for each side without changing the numerical code inside the solver.

When assembling the matrix $[K]$ and vector $\{rhs\}$, the key idea is to recognize that **different nodes require different stencil formulas** depending on their position in the domain. At every iteration of the (i, j) loop you should:

1. Compute the linear index k associated with node (i, j).
2. Determine whether the node is:
   - a **corner** (S–W, S–E, N–W, N–E),
   - an **edge** node (South, West, East, North),
   - or an **interior** node.
3. Apply the appropriate finite-difference stencil. Check out the slides we have discussed in class (or the class notes) for the different nodal expressions.

A compact way to visualize this logic is:

```
for j = 1:ny
    for i = 1:nx
        k = linear_index(i,j);

        if is_corner(i,j)
            % special stencil for corner
        elseif is_edge(i,k)
            % stencil for edges (Dirichlet or Neumann)
        else
            % interior stencil
        end

    end
end
```

- Create a new function `FDM_2D_s2` and use it to solve this problem involving Dirichlet and Neumann BCs

$$
\begin{cases}
\dfrac{\partial^2 \varphi}{\partial x^2} + \dfrac{\partial^2 \varphi}{\partial y^2} - 1 = 0, & \\
\qquad \varphi(x, 0) = 1 & \text{South edge,} \\
\qquad \varphi(0, y) = -1 & \text{West edge,} \\
\qquad \left.\dfrac{\partial \varphi}{\partial x}\right|_E = 0 & \text{East edge,} \\
\qquad \left.\dfrac{\partial \varphi}{\partial y}\right|_N = 0 & \text{North edge.}
\end{cases}
\tag{3}
$$

with the following settings: $n_x = 25; n_y = 25$.
According to the formulation the BC structure should be set to:

```
BC.S.type = 'D';
BC.S.val = 1;

BC.W.type = 'D';
BC.W.val = -1;

BC.E.type = 'N';
BC.E.val = 0;

BC.N.type = 'N';
BC.N.val = 0;
```

Run the code and check your solution, which should look like the following:

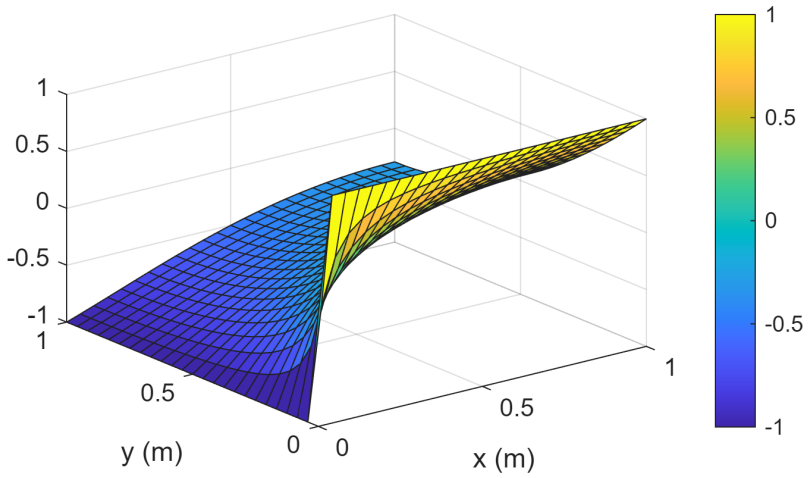


Figure 3: Solution with mixed Dirichlet/Neumann BCs and `t=1`