Finite difference method for the Poisson problem in 2D

November 22, 2021

1 The finite difference method for the Poisson problem in 2D

Note: The code parts of this notebook won't be relevant for the exam!

[1]:
```
from IPython.core.display import HTML
def css_styling():
    try:
        fname = "calculus4N.css"
        with open(fname, "r") as f:
            styles = f.read()
            return HTML(styles)
    except FileNotFoundError:
        print(f"Could not find file {fname}!")

# Comment out next line and execute this cell to restore the default notebook␣
↪style
css_styling()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.1 The Poisson Equation in 2D

We consider 2-dimenionsal equivalent of the two-point boundary value problem, known as the **Poisson problem**:

Let $\Omega = [0,1] \times [0,1] \subset \mathbb{R}^2$, and given a right-hand side (or source) function $f : \Omega \to \mathbb{R}$ and a boundary function $g : \partial\Omega \to \mathbb{R}$. Here $\partial\Omega = \{0\} \times [0,1] \cup \{1\} \times [0,1] \cup [0,1] \times \{0\} \cup [0,1] \times \{1\}$ denotes the boudary of Ω. Then the task is to find $u : \Omega \to \mathbb{R}$ such that

$$-\Delta u = f \quad \text{in } \Omega, \tag{1a}$$

$$u = g \quad \text{on } \partial\Omega. \tag{1b}$$

Recall that the Laplace operater Δu is defined by

$$\Delta u(x,y) = \partial_x^2 u(x,y) + \partial_y^2 u(x,y) = \frac{\partial^2}{\partial x^2} u(x,y) + \frac{\partial^2}{\partial y^2} u(x,y)$$

How do we compute a numerical solution to (1a)-(1b)?

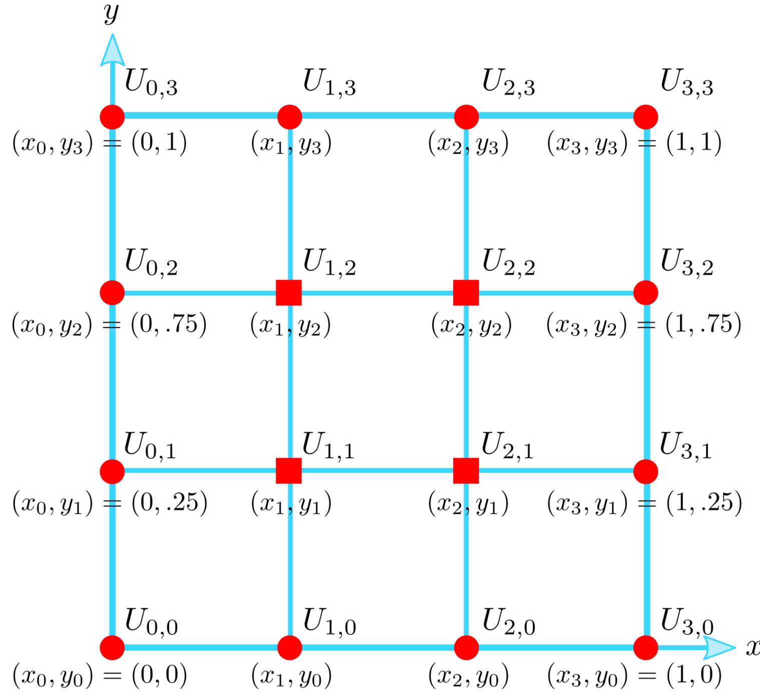
1.2 Finite Difference Method for the 2D Poisson problem

Instead of trying to compute $u(x)$ exactly, we will now try to compute a numerical approximation u_{Δ} of $u(x)$. In 1D, we introduced $n+1$ equally space grid points on $[0,1]$. Since we are in 2D now, we just apply the same procedure in every dimension and then create a 2D grid:

- Subdivide the x-axis, and introduce $\{x_i\}_{i=0}^n$ with $x_i = ih$, $h = \frac{1}{n}$
- Subdivide the y-axis, and introduce $\{y_j\}_{j=0}^n$ with $y_j = jh$
- Defind the $N = (n+1)^2$ grid points $\{(x_i, y_j)\}_{i,j=0}^n$.

To each of the grid points (x_i, y_j) we now assoicate an unknow $U_{i,j}$ for $i,j=0,…n $.

Below you see an illustration for the case $n = 3$:



To derive an equation system for the $U_{i,j}$, we take the same approach as for the two-point value problem realizing that the $\partial_x^2 u$ can be approximated by a central difference operator along the x-axis

$$\partial_x^+ \partial_x^- u(x_i, y_j) := \frac{u(x_{i+1}, y_j) - 2u(x_i, y_j) + u(x_{i-1}, y_j)}{h^2} \approx \partial_x^2 u(x_i, y_j),$$

while keeping the y-variable fixed.

The same goes the other way around, so to approximate $\partial_y^2 u$ at (x_i, y_j), we use the central difference operator along the y-axis

$$\partial_y^+ \partial_y^- u(x_i, y_j) := \frac{u(x_i, y_{j+1}) - 2u(x_i, y_j) + u(x_i, y_{j-1})}{h^2} \approx \partial_y^2 u(x_i, y_j),$$

while keeping the x-variable fixed.

So in total, we obtain that

$$
\begin{aligned}
f(x_i, y_i) &= -\Delta u(x_i, y_i) \\
&\approx -\partial_x^+ \partial_x^- u(x_i, y_j) - \partial_y^+ \partial_y^- u(x_i, y_j) \\
&= -\frac{u(x_{i+1}, y_j) + u(x_i, y_{j+1}) - 4u(x_i, y_j) + u(x_{i-1}, y_j) + u(x_i, y_{j-1})}{h^2}
\end{aligned}
$$

Because of the index structure the finite difference operator $(\partial_x^+ \partial_x^- + \partial_y^+ \partial_y^-)$ is also called **5-point stencil**.

1.2.1 Exercise 1

Similar as before, use Taylor expansion to show that for $u \in C^4([0,1]^2)$

$$
\max_{(x,y) \in [0,1]^2} |(\partial_x^+ \partial_x^- + \partial_y^+ \partial_y^-) u(x) - \Delta u(x,y)| = \mathcal{O}(h^2).
$$

Using the 5-point stencil, we again get an equation system for the $(N-1)^2$ **internal grid points** $\{(x_i, y_j\}_{i,j=1}^{n-1}$

$$
\begin{aligned}
-(\partial_x^+ \partial_x^- + \partial_y^+ \partial_y^-) U_{ij} &= \frac{4U_{i,j} - U_{i+1,j} - U_{i,j+1} - U_{i-1,j} - U_{i,j-1}}{h^2} && (1) \\
&= f(x_i, y_j) =: F_{ij} \quad \text{for } i, j = 1, \ldots N-1. && (2)
\end{aligned}
$$

As before (yes, we are repeating ourselves!) the system needs to closed by supplementing the equations for the boundary conditions. We set the boundary conditions on the bottom and top of the square $[0,1]^2$ by requiring that

$$
U_{i,j} = g(x_i, y_j) \quad \text{for } i = 0, \ldots, n, j \in \{0, n\}. \tag{3}
$$

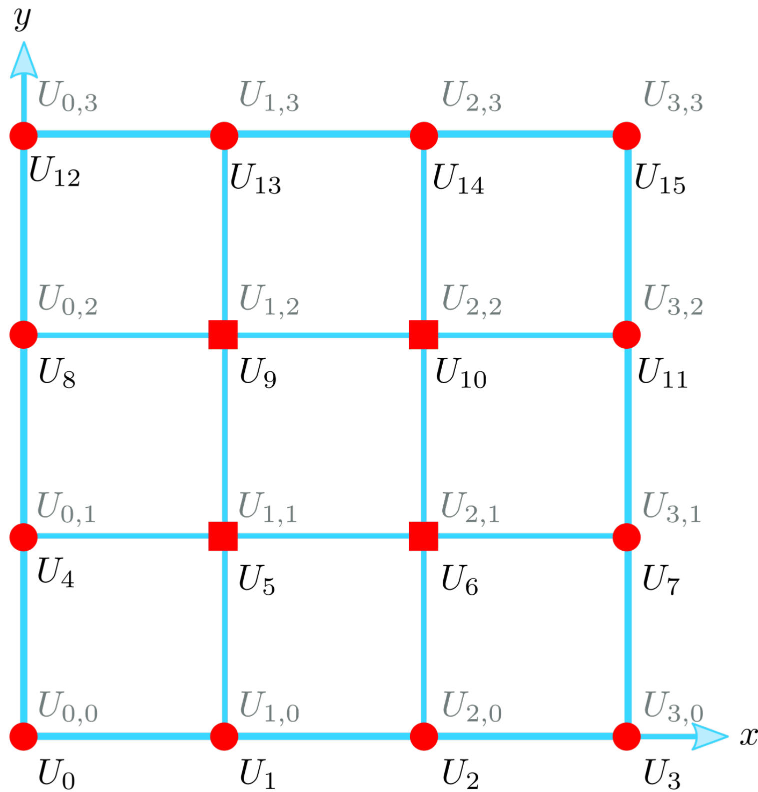
and then treating the remaining boundary points on the left and right of $[0,1]^2$:

$$
U_{i,j} = g(x_i, y_j) \quad \text{for } i \in \{0, n\}, j = 1, \ldots, n-1. \tag{4}
$$

How can we get from here to a nice looking linear system? We have used a double index, one for each dimension, so that we could easily reduce the discretization of Δ to the techniques we learned in Chapter 1 on 1D two-point boundary problems.

To avoid the introduction of multi-dimensional matrices, we need to transform the double index (i, j) into a single index by introducing a consecutive numbering $I = I(i, j)$ of the the unknowns.

For example, the row-wise numbering of the unknown is illustrated here:



1.2.2 Exercise 2:

Any consecutive numbering is nothing but a index mapping of the form $\mathbb{N}^2 \ni (i,j) \mapsto I(i,j) \in \mathbb{N}$. Which of the following index mapping corresponds to the row-wise numbering illustrated above? 1. $I(i,j) = in + j \quad$ for $i,j = 0,\dots,n$ 2. $I(i,j) = i + jn \quad$ for $i,j = 0,\dots,n$ 3. $I(i,j) = i + j(n+1)$ for $i,j = 0,\dots,n$ 4. $I(i,j) = i(n+1) + j \quad$ for $i,j = 0,\dots,n$

Solution:

1.2.3 Exercise 3

Solution:
A row-wise numbering is given by 3. A column-wise numbering is given by 4.

1.3 Implementation of a first FDM solver in 2D

Start with defining a 1-line function `I(i,j,n)` which for n equally spaced intervals in each direction transforms an double index (i,j) into a single index I using a row-wise numbering.

[4]:
```
# Define index mapping
def I(i,j,n):
    return i + j*(n+1)
```

Next, define a `def fdm_poisson_2d_matrix_dense(n, I)` which takes in n and the index mapping I and computes the full finite difference matrix, including setting those diagonals elements to 1 which correspond to an index on the boundary.

[5]:
```
import numpy as np

def fdm_poisson_2d_matrix_dense(n, I):
    # Gridsize
    h = 1.0/n

    # Total number of unknowns is N = (n+1)*(n+1)
    N = (n+1)**2

    # Define zero matrix A of right size and insert 0
    A = np.zeros((N,N))

    # Define tridiagonal part of A
    hh = h*h
    for i in range(1, n):
        for j in range(1, n):
            Iij = I(i,j,n)
            A[Iij,Iij] = 4/hh   # U_ij
            A[Iij,I(i-1,j,n)] = -1/hh  # U_{i-1,j}
            A[Iij,I(i+1,j,n)] = -1/hh  # U_{i+1,j}
            A[Iij,I(i,j-1,n)] = -1/hh  # U_{i,j-1}
            A[Iij,I(i,j+1,n)] = -1/hh  # U_{i,j+1}

    # Incorporate boundary conditions
    # Add boundary values related to unknowns from the first and last grid ROW
    for j in [0,n]:
        for i in range(0,n+1):
            Iij = I(i,j,n)
            A[Iij, Iij] = 1 # U_ij

    # Add boundary values related to unknowns from the first and last grid COLUMN
    for i in [0,n]:
        for j in range(0,n+1):
        # Note we set corner points twice, change to range(1,N) to avoid this
            Iij = I(i,j,n)
            A[Iij, Iij] = 1 # U_ij

    return A
```

Now try to numerically solve the Poisson problem. We will learn a few new functions from the numpy module along the way.

[6]:
```
# Define subdivions in each direction
n = 10

# To define the grid we use "linspace" in each direction ...
xi = np.linspace(0,1,n+1)
yi = np.linspace(0,1,n+1)
```

```
# ... and then generate a grid very similar to the illustration above
# The default value for the meshgrid parameter sparse is "False"
# see https://docs.scipy.org/doc/numpy/reference/generated/numpy.meshgrid.html
# but in principal we only need to now xi yi
# so to save memory you can store as in sparse format. Try both out and
# print the x and y results to see what you get.
x,y = np.meshgrid(xi,yi,sparse=True)
print(x)
print(y)
```

```
[[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]]
[[0. ]
 [0.1]
 [0.2]
 [0.3]
 [0.4]
 [0.5]
 [0.6]
 [0.7]
 [0.8]
 [0.9]
 [1. ]]
```

To build a complete example and test your code, we use again the method of **manufactured solution**.

[7]:
```
# Example of exact solution
def u_ex(x, y):
    return np.sin(1*np.pi*x)*np.sin(2*np.pi*y)

# Boundary data g is given by u_ex
g = u_ex

# Right hand side
def f(x, y):
    return ((1*np.pi)**2 + (2*np.pi)**2)*np.sin(1*np.pi*x)*np.sin(2*np.pi*y)

# Evaluate u on the grid. The output will be a 2 dimensional array
# where U_ex_grid[i,j] = u_ex(x_i, y_j)
U_ex_grid = u_ex(x,y)

# Print f_grid  to see how it looks like!
print(U_ex_grid)
```

```
[[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.81635632e-01  3.45491503e-01  4.75528258e-01
```

```
 5.59016994e-01  5.87785252e-01  5.59016994e-01  4.75528258e-01
 3.45491503e-01  1.81635632e-01  7.19829328e-17]
[ 0.00000000e+00  2.93892626e-01  5.59016994e-01  7.69420884e-01
 9.04508497e-01  9.51056516e-01  9.04508497e-01  7.69420884e-01
 5.59016994e-01  2.93892626e-01  1.16470832e-16]
[ 0.00000000e+00  2.93892626e-01  5.59016994e-01  7.69420884e-01
 9.04508497e-01  9.51056516e-01  9.04508497e-01  7.69420884e-01
 5.59016994e-01  2.93892626e-01  1.16470832e-16]
[ 0.00000000e+00  1.81635632e-01  3.45491503e-01  4.75528258e-01
 5.59016994e-01  5.87785252e-01  5.59016994e-01  4.75528258e-01
 3.45491503e-01  1.81635632e-01  7.19829328e-17]
[ 0.00000000e+00  3.78436673e-17  7.19829328e-17  9.90760073e-17
 1.16470832e-16  1.22464680e-16  1.16470832e-16  9.90760073e-17
 7.19829328e-17  3.78436673e-17  1.49975978e-32]
[-0.00000000e+00 -1.81635632e-01 -3.45491503e-01 -4.75528258e-01
-5.59016994e-01 -5.87785252e-01 -5.59016994e-01 -4.75528258e-01
-3.45491503e-01 -1.81635632e-01 -7.19829328e-17]
[-0.00000000e+00 -2.93892626e-01 -5.59016994e-01 -7.69420884e-01
-9.04508497e-01 -9.51056516e-01 -9.04508497e-01 -7.69420884e-01
-5.59016994e-01 -2.93892626e-01 -1.16470832e-16]
[-0.00000000e+00 -2.93892626e-01 -5.59016994e-01 -7.69420884e-01
-9.04508497e-01 -9.51056516e-01 -9.04508497e-01 -7.69420884e-01
-5.59016994e-01 -2.93892626e-01 -1.16470832e-16]
[-0.00000000e+00 -1.81635632e-01 -3.45491503e-01 -4.75528258e-01
-5.59016994e-01 -5.87785252e-01 -5.59016994e-01 -4.75528258e-01
-3.45491503e-01 -1.81635632e-01 -7.19829328e-17]
[-0.00000000e+00 -7.56873346e-17 -1.43965866e-16 -1.98152015e-16
-2.32941664e-16 -2.44929360e-16 -2.32941664e-16 -1.98152015e-16
-1.43965866e-16 -7.56873346e-17 -2.99951957e-32]]
```

Here is a little helper functions for plotting grid functions like `U_grid`.

[8]:
```python
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

def plot2D(X, Y, Z, title=""):
    # Define a new figure with given size an
    fig = plt.figure(figsize=(8, 6), dpi=100)
    ax = fig.gca(projection='3d')
    surf = ax.plot_surface(X, Y, Z,
                           rstride=1, cstride=1, # Sampling rates for the x and␣
 ↪y input data
                           cmap=cm.viridis)      # Use the new fancy colormap␣
 ↪viridis

    # Set initial view angle
    ax.view_init(30, 225)
```

```
    # Set labels and show figure
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_title(title)
    plt.show()
```

Now try it out to plot u_{ex}.

[9]:
```
%matplotlib widget

plot2D(x, y, U_ex_grid, title="$u(x,y)$")
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', 'home'), ('Back', 'Bac
```

You can do the same for right-hand side f and the boundary function g.

[10]:
```
# Evaluate f on the grid. The output will be a 2 dimensional array
# where f_grid[i,j] = f(x_i, y_j)
F_grid = f(x,y)

# Same game for boundary data g
G_grid = g(x,y)
```

Before we finally going to solve the Poisson problem, we need translate the `F_grid` into a proper rhs vector F and need to incorporate the boundary function value into F.

Start with flatten out F and G:

[11]:
```
# To apply bcs we have to flatten out F which is done by the ravel function
F = F_grid.ravel()
print(F)

# To apply bcs we have to flatten out G which is done by the ravel function
G = G_grid.ravel()
```

```
[ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
  8.96335916e+00  1.70493223e+01  2.34663789e+01  2.75863829e+01
  2.90060396e+01  2.75863829e+01  2.34663789e+01  1.70493223e+01
  8.96335916e+00  3.55221535e-15  0.00000000e+00  1.45030198e+01
  2.75863829e+01  3.79693987e+01  4.46357052e+01  4.69327579e+01
  4.46357052e+01  3.79693987e+01  2.75863829e+01  1.45030198e+01
  5.74760517e-15  0.00000000e+00  1.45030198e+01  2.75863829e+01
  3.79693987e+01  4.46357052e+01  4.69327579e+01  4.46357052e+01
  3.79693987e+01  2.75863829e+01  1.45030198e+01  5.74760517e-15
  0.00000000e+00  8.96335916e+00  1.70493223e+01  2.34663789e+01
  2.75863829e+01  2.90060396e+01  2.75863829e+01  2.34663789e+01
```

```
 1.70493223e+01  8.96335916e+00  3.55221535e-15  0.00000000e+00
 1.86751013e-15  3.55221535e-15  4.88920499e-15  5.74760517e-15
 6.04338972e-15  5.74760517e-15  4.88920499e-15  3.55221535e-15
 1.86751013e-15  7.40101788e-31 -0.00000000e+00 -8.96335916e+00
-1.70493223e+01 -2.34663789e+01 -2.75863829e+01 -2.90060396e+01
-2.75863829e+01 -2.34663789e+01 -1.70493223e+01 -8.96335916e+00
-3.55221535e-15 -0.00000000e+00 -1.45030198e+01 -2.75863829e+01
-3.79693987e+01 -4.46357052e+01 -4.69327579e+01 -4.46357052e+01
-3.79693987e+01 -2.75863829e+01 -1.45030198e+01 -5.74760517e-15
-0.00000000e+00 -1.45030198e+01 -2.75863829e+01 -3.79693987e+01
-4.46357052e+01 -4.69327579e+01 -4.46357052e+01 -3.79693987e+01
-2.75863829e+01 -1.45030198e+01 -5.74760517e-15 -0.00000000e+00
-8.96335916e+00 -1.70493223e+01 -2.34663789e+01 -2.75863829e+01
-2.90060396e+01 -2.75863829e+01 -2.34663789e+01 -1.70493223e+01
-8.96335916e+00 -3.55221535e-15 -0.00000000e+00 -3.73502025e-15
-7.10443070e-15 -9.77840997e-15 -1.14952103e-14 -1.20867794e-14
-1.14952103e-14 -9.77840997e-15 -7.10443070e-15 -3.73502025e-15
-1.48020358e-30]
```

Also, we define a function incorporating the values of G into F.

[12]:
```python
def apply_bcs(F, G, n, I):
    # Add boundary values related to unknowns from the first and last grid ROW
    bc_indices = [ I(i,j,n)  for j in [0, n] for i in range(0, n+1) ]
    F[bc_indices] = G[bc_indices]

    # Add boundary values related to unknowns from the first and last grid COLUMN
    bc_indices = [ I(i,j,n) for i in [0, n] for j in range(0, n+1) ]
    F[bc_indices] = G[bc_indices]
```

Finally, solve the Poisson problem.

[13]:
```python
# Linear algebra solvers from scipy
import scipy.linalg as la

# Compute the FDM matrix
A = fdm_poisson_2d_matrix_dense(n, I)

# Apply bcs
apply_bcs(F, G, n, I)

# Solve
U = la.solve(A, F)

# Make U into a grid function for plotting
U_grid = U.reshape((n+1,n+1))
```

[14]:
```
%matplotlib widget
# and plot u
plot2D(x, y, U_grid, title="$u(x,y)$")
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', 'home'), ('Back', 'Bac
```

1.4 Convergence properties of the finite difference method in 2D

Next, we use manufactured analytical reference solution u_{ex} to compute the experimental order of convergence (EOC) for $n = 16, 32, 64$ using $\max_{i,j} |U_{ij} - u(x_i, y_j)|$ as error measure.

[15]:
```
def fdm_poisson_2d_solver_dense(F, G, n, I):

    # Compute the FDM matrix
    A = fdm_poisson_2d_matrix_dense(n, I)

    # Apply bcs
    apply_bcs(F, G, n, I)

    # Solve
    return la.solve(A, F)
```

[16]:
```
def convergence_test(u_ex, f, g, poisson_solver, ns):

    # Store x, y, U and err as tuples here
    data = []
    for n in ns:
        xi = np.linspace(0, 1, n+1)
        yi = np.linspace(0, 1, n+1)
        x, y = np.meshgrid(xi, yi, sparse=True)

        U_ex_grid = u_ex(x,y)
        G_grid = u_ex(x,y)
        F_grid = f(x,y)

        U = poisson_solver(F_grid.ravel(), G_grid.ravel(), n, I)
        err = np.abs(U_ex_grid.ravel() - U).max()
        print("max |U_ex-U| = {}".format(err))

        data.append((x,y,U,n,err))

    return data
```

[17]:
```
import pandas as pd
from IPython.display import display

# Run convergence test
```

```
ns = [8, 16, 32, 64]
data = convergence_test(u_ex, f, g, fdm_poisson_2d_solver_dense, ns)

# Compute eoc
ns = np.array(ns)
errs = np.array([data_n[-1] for data_n in data] )
eocs = np.log(errs[1:]/errs[:-1])/np.log(ns[:-1]/ns[1:])
eocs = np.insert(eocs, 0, np.infty)

eoc_table = pd.DataFrame({'N': ns, 'EOC': eocs })
display(eoc_table)
```

```
max |U_ex-U| = 0.04476185092458085
max |U_ex-U| = 0.010989314920720306
max |U_ex-U| = 0.00273495483262276
max |U_ex-U| = 0.0006829683944282738
```

	N	EOC
0	8	inf
1	16	2.026168
2	32	2.006513
3	64	2.001626

[19]:
```
# Plot last solution
x, y, U, n, errs = data[-1]
U_grid = U.reshape((n+1,n+1))

plot2D(x, y, U_grid, title="$u(x,y)$")
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', 'home'), ('Back', 'Bac
```

As in the 1D case we observe 2nd order convergence with respect to the maximum norm. This can be proven theoretically.

1.5 Theorem

If $u \in C^4([0,1]^2)$ then finite difference method is **2nd order convergent** with respect to the maximums norms, that is,

$$\max_{i,j} |U_{ij} - u(x_i, y_j)| \leqslant Ch^2 |u|_{C^4} = \mathcal{O}(h^2).$$

Proof. Omitted.

1.6 Sparse Matrices

In this section we will have a quick look at what happens if we solve the Poisson problem in 2D for a reasonably large number of grid points.

Note: The remaining material will not be considered for the exam!

1.6.1 A numerical scaling experiment

Test how large you can chose the resolution n until either the problem takes too long (say 2 minutes) to compute or uses too much memory.

To do so you can use `%timeit` and `%%timeit` magic functions in IPython, see corresponding documentation.

In a nutshell, `%%timeit` measures the executation time of an entire cell, while `%timeit` only measures only the executation time of a single line, e.g. as in

```
%timeit my_function()
```

Can you explain why the problem scales so badly with respect to number of unknowns $N = (n+1)^2$?

Regarding the usage of `timeit`: To obtain reliable timeings, `timeit` does not perform a single run, but rather a number of runs , and in each run, the given statement is executed times in a loop. This can sometimes lead to large waiting time, so you can change that by time

```
%timeit -r <R> -n <N> my_function()
```

Also if you want to store the value of the best run by passing the option `-o`:

```
timings_data = %timeit -o my_function()
```

which stores the data from the timing experiment. You can access the best time measured in seconds by

```
timings_data.best
```

Example:

[20]:
```
n = 10

xi = np.linspace(0, 1, n+1)
yi = np.linspace(0, 1, n+1)
x, y = np.meshgrid(xi, yi, sparse=True)

G_grid = u_ex(x,y)
F_grid = f(x,y)

data = %timeit -o -r 1 -n 1 fdm_poisson_2d_solver_dense(F_grid.ravel(), G_grid.
 ↪ravel(), n, I)
print(data)
print(f"Best timing: {data.best}")
```

```
651 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
651 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
Best timing: 0.0006514100023196079
```

Discussion:

In the experiment, we see that whenever we double the number n of grid points in each direction, the overall computation time increases with roughly a factor of

As the number of total unknowns $N = (n+1)^2$ scales like n^2 (modulo lower order terms) we would rather expect that the computation time increases with a factor of $4 = 2^2$ whenever we double the number of grid points in each space direction.

There are several reasons for the bad scaling, but the main reason is that our current matrix data structures and the linear algebra solver do not take into account that we the resulting finite difference matrix is **sparse**.

The term sparsity refers to the fact that while the matrix is of size $N \times N = (n+1)^2 \times (n+1)^2$, only roughly $\mathcal{O}(N)$ entries are non-zero. This is because an unknown U_i only interacts with at most 4 other unknowns through the 5-point stencil.

Since our algorithms and data structures do not take into account the sparse structure, we run into two problems:

- Waste of memory: A standard matrix implementation will reserve memory for any single of the $\mathcal{O}(N^2) = \mathcal{O}(n^4)$ element of the Matrix A, while in principle we only need to store roughly $\mathcal{O}(N) = \mathcal{O}(n^2)$ elements!
- Waste of computational time: The computational time of standard LU solver based on Gaussian elimination techniques scales like $\mathcal{O}(N^3) = \mathcal{O}(n^6)$. That means if we double n then the solver time will increase by a factor of $2^6 = 64$!

As a remedy, many data structures and solution algorithms have been developed specifically with sparse matrices in mind. Unfortunately, we don't have time to delve into the exciting topic, but will just use the sparse equivalents of the data structures and solvers available in the `scipy` module.

1.6.2 Exercise 4

Based on your implementation above, we now implement an improved finite difference solver using **sparse matrices**. Sparse matrices only store the non-zero elements of a matrix. Note that the number of non-zero elements in the finite difference matrix scales like N and not like N^2 like **full matrices**.

Knowing the structure and entries of the matrix a priori, the most efficient realization would be based on (block) tridiagonal sparse matrices. But to allow for minimal adjustments of your previous solver implementation, we simply switch to a flexible sparse matrix format

To this end you have change only 3 lines of code and incorporate the following code snippets into your previous code. For comparision you may want to define a separate function `fdm_poisson_2d_matrix_sparse(n, I)`.

1.6.3 Code Snippets

Get access to sparse matrices and sparse solvers.

```
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve
```

Use a sparse matrix format for A, see https://docs.scipy.org/doc/scipy/reference/sparse.html for the many formats which are available. Here we use "dictionary of keys" based representation which is an empty matrix to begin with and which can easily filled with non-zero values at the appropriate places.

```
A = sp.dok_matrix((N, N))
```

After creating the matrix we have to convert it to a different format, the so-called "Compressed Sparse Row matrix" representation, which is much more efficient when solving the system $AU = F$ with a sparse solver. see https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.spsolve.html

```
A_csr = A.tocsr()
U = spsolve(A_csr, F)
```

Solution:

[21]:
```
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve

def fdm_poisson_2d_matrix_sparse(n, I):
    # Gridsize
    h = 1.0/n

    # Total number of unknowns is N = (n+1)*(n+1)
    N = (n+1)**2

    # Define zero sparse matrix A
    A = sp.dok_matrix((N, N))

    # Define tridiagonal part of A
    hh = h*h
    for i in range(1, n):
        for j in range(1, n):
            Iij = I(i,j,n)
            A[Iij,Iij] = 4/hh    # U_ij
            A[Iij,I(i-1,j,n)] = -1/hh  # U_{i-1,j}
            A[Iij,I(i+1,j,n)] = -1/hh  # U_{i+1,j}
            A[Iij,I(i,j-1,n)] = -1/hh  # U_{i,j-1}
            A[Iij,I(i,j+1,n)] = -1/hh  # U_{i,j+1}

    # Incorporate boundary conditions
    # Add boundary values related to unknowns from the first and last grid ROW
    for j in [0,n]:
        for i in range(0,n+1):
            Iij = I(i,j,n)
            A[Iij, Iij] = 1 # U_ij

    # Add boundary values related to unknowns from the first and last grid COLUMN
```

```python
    for i in [0,n]:
        for j in range(0,n+1):
        # Note we set corner points twice, change to range(1,N) to avoid this
            Iij = I(i,j,n)
            A[Iij, Iij] = 1 # U_ij

    return A
```

[22]:
```python
def fdm_poisson_2d_solver_sparse(F, G, n, I):

    # Compute the FDM matrix
    A = fdm_poisson_2d_matrix_sparse(n, I)

    # Apply bcs
    apply_bcs(F, G, n, I)

    # Solve
    A_csr = A.tocsr()
    return solve(A_csr, F)
```

[23]:
```python
# Run convergence test
ns = [8, 16, 32, 64]
data = convergence_test(u_ex, f, g, fdm_poisson_2d_solver_dense, ns)

# Plot last solution
x, y, U, n, errs = data[-1]
U_grid = U.reshape((n+1,n+1))
plot2D(x, y, U_grid, title="$u(x,y)$")

# Compute eoc
ns = np.array(ns)
errs = np.array([data_n[-1] for data_n in data] )
eocs = np.log(errs[1:]/errs[:-1])/np.log(ns[:-1]/ns[1:])
print(f"EOCs = {eocs}")
```

```
max |U_ex-U| = 0.04476185092458085
max |U_ex-U| = 0.010989314920720306
max |U_ex-U| = 0.00273495483262276
max |U_ex-U| = 0.0006829683944282738
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', 'home'), ('Back', 'Bac
```

```
EOCs = [2.02616824 2.00651254 2.00162629]
```

1.6.4 Exercise 5

Measure and compare the overall solution time for your two implementations 'fdm_poisson_2d_dense' and 'fdm_poisson_2d_sparse' for a number of grid sizes $h = 1/N$

using the cell magic command `%%timeit` of `%timeit`.