Print to PDF

Lecture 6

Contents

In lecture 6 we will move from one to two spatial dimensions and discuss how to discretize and solve a boundary value problem (Poisson’s equation). Important topics are

- The Cartesian product - definition and the use for computational meshes
- The finite difference method in 2D for the Laplace operator
- The vec-trick (also referred to as vectorization, but not the same as Numpy vectorization)
- The Kronecker product

Two-dimensional spatial domain

We will now consider equations within a simple two-dimensional rectangular domain. To get started we will look at the Poisson equation

$$\nabla^2 u = f,$$

which in two-dimensional Cartesian coordinates is

Skip to main content

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f.$$

We consider the equation for a rectangular domain

$$(x, y) \in [0, L_x] \times [0, L_y],$$

and Dirichlet boundary conditions. A Dirichlet boundary condition means that the solution is known (or fixed) at the boundary. There are now 4 sides to a rectangle and the solutions needs to be fixed at all these boundaries since there are two derivatives in both x and y directions. We need to fix:

$$u(x=0, y), \quad u(x=L_x, y)$$

$$u(x, y=0), \quad u(x, y=L_y)$$

In the figure below the locations of these Dirichlet conditions are plotted in color.

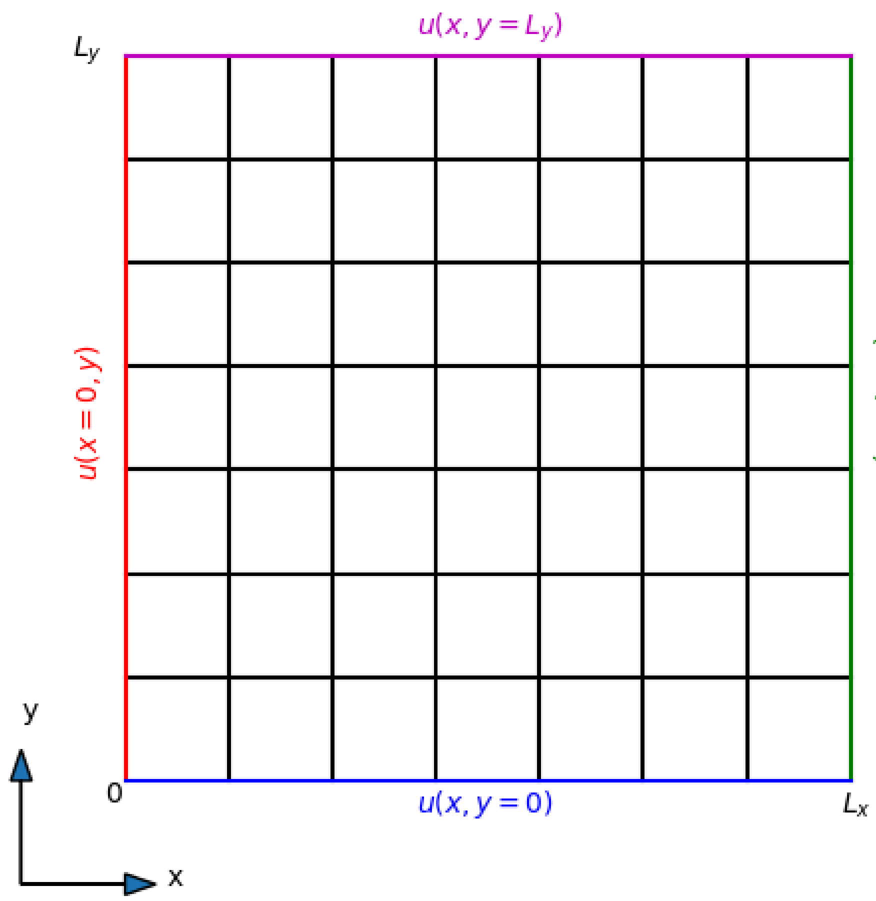
Note that the conditions at the two sides $x=0$ and $x=L_x$ can be functions of the other coordinate y. Likewise, for the two boundaries with constant y the condition may be a function of x. Also note that the boundary conditions need to be consistent in the corners, because each corner belongs to two sides.

We will at first simply use a constant value 0 along all boundaries since this is easiest. In that case

$$u(x=0, y) = u(x=L_x, y) = u(x, y=0) = u(x, y=L_y) = 0$$

▶ Show code cell source

Skip to main content



Discretization

We discretize the mesh using two lines, one in the x-direction and one in the y-direction:

$$x_i = i\Delta x, \quad i = 0, 1, \ldots, N_x$$

$$y_j = j\Delta y, \quad j = 0, 1, \ldots, N_y$$

where $\Delta x = L_x/N_x$ and $\Delta y = L_y/N_y$. The mesh is plotted above for $N_x = N_y = 8$. We will use vector notation for the mesh, with

$$\boldsymbol{x} = (x_0, x_1, \ldots, x_{N_x})$$

$$\boldsymbol{y} = (y_0, y_1, \ldots, y_{N_y})$$

Skip to main content

Cartesian products

In order to create a two-dimensional mesh, like the one seen above, we use a Cartesian product. The Cartesian product is most easily explained through an example. Consider the two arrays (or here lists) $\boldsymbol{u} = [1, 2, 3]$ and $\boldsymbol{v} = [4, 5]$. We can compute the Cartesian product of these two arrays using the Python package itertools

```python
import itertools
u = [1, 2, 3]
v = [4, 5]
uxv = itertools.product(u, v)
list(uxv)
```

```
[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
```

So we get a list containing all possible pairs of items, one from $\boldsymbol{u}$ and one from $\boldsymbol{v}$. Since there are 3 items in $\boldsymbol{u}$ and 2 in $\boldsymbol{v}$ we get a total of $2 \cdot 3 = 6$ pairs of numbers.

The Cartesian product $\boldsymbol{u} \times \boldsymbol{v}$ can be described mathematically as

$$\boldsymbol{u} \times \boldsymbol{v} = \{(u, v) \,|\, u \in \boldsymbol{u} \text{ and } v \in \boldsymbol{v}\},$$

which reads all the pairs (u, v) such that u is in the set $\boldsymbol{u}$ and v in the set $\boldsymbol{v}$.

What does this have to do with a discretization of a computational mesh? Well the computational mesh is a combination of all pairs (x, y) such that x is in the 1D mesh along the x-direction $(0, \Delta x, 2\Delta x, \ldots)$ and y is in the 1D mesh along the y-direction ($0, \Delta y, 2\Delta y, \ldots$). We have already defined these two meshes above in the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. So the computational mesh is really

$$\boldsymbol{x} \times \boldsymbol{y} = \{(x, y) \,|\, x \in \boldsymbol{x} \text{ and } y \in \boldsymbol{y}\},$$

all pairs (x, y) such that x is in $\boldsymbol{x}$ and y is in $\boldsymbol{y}$.

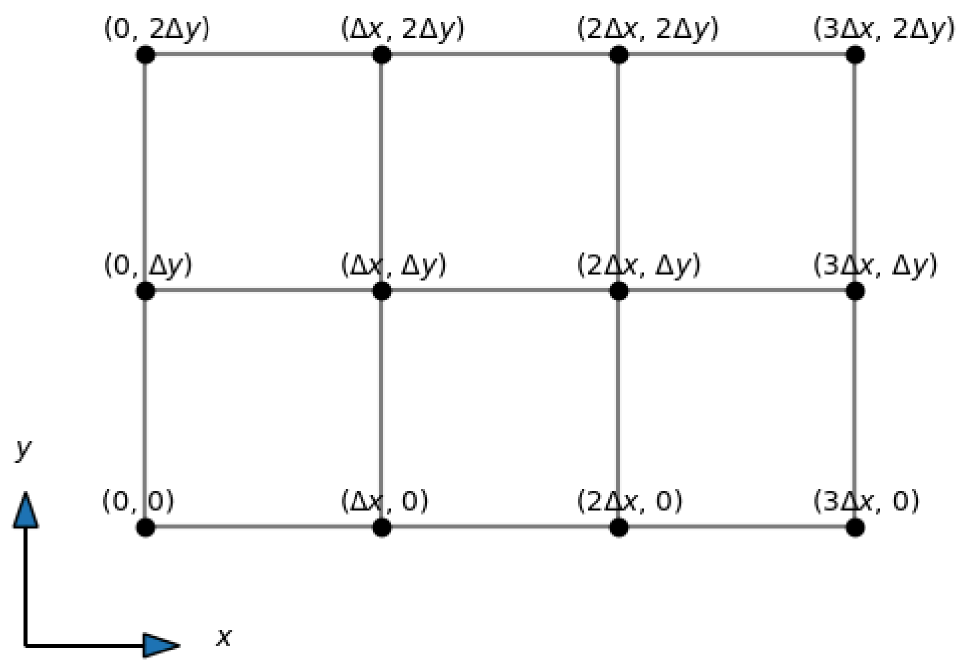
If the meshes are simply

$$\boldsymbol{x} = (0, \Delta x, 2\Delta x, 3\Delta x) \text{ and } \boldsymbol{y} = (0, \Delta y, 2\Delta y),$$

then a two-dimensional mesh would be all possible combinations of these arrays, as shown below.

Skip to main content

▶ Show code cell source



Note that there are $4 \cdot 3 = 12$ pairs of numbers. If we drop Δx and Δy from the pairs and use only the integers in front of them, we get with itertools:

```python
xy = list(itertools.product([0, 1, 2, 3], [0, 1, 2]))
print(xy)
```

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3,
```

We see that itertools (or the Cartesian product) presents the result as an array of pairs of length 12 (12 pairs). It does not shape the product into a Cartesian two-dimensional mesh. The Cartesian product is the same as a double for loop

```python
print([(a, b) for a in range(4) for b in range(3)])
```

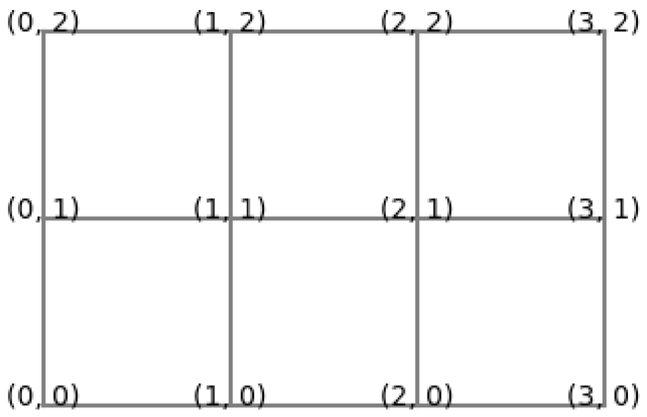
```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3,
```

However, when we illustrate these numbers in a Cartesian mesh, then we see that it is only

Skip to main content

```python
fig = plt.figure(figsize=(4, 4))
for i in range(3):
    plt.plot((0, 3), (i, i), 'grey')
for j in range(4):
    plt.plot((j, j), (0, 2), 'grey')
for m in xy:
    plt.text(m[0], m[1], f'({m[0]}, {m[1]})', horizontalalignment='center')

ax = fig.gca()
ax.axis('off')
ax.axis('equal');
```



Note that if this mesh was a two-dimensional matrix, then i and j in (i, j) would represent column and row, respectively, which is counterintuitive. We normally think about a matrix A with items a_{ij}, where the first index i represents row and the second index j represents column. This is a complicating factor for the Cartesian mesh that is important to be aware of. But it is not a mistake. The numbers in the plot above are not indices of a matrix, they represent (x, y) in a Cartesian mesh.

meshgrid

The Cartesian mesh gives both x and y coordinates for the entire two-dimensional mesh and it is created from a Cartesian product of one-dimensional meshes along x and y-directions. Luckily for us Numpy has a function that computes this mesh for us and we do not have to use itertools directly.

Let us now compute the Cartesian mesh for

Skip to main content

$$\boldsymbol{x} = (0, 1, 2, 3) \quad \text{and} \quad \boldsymbol{y} = (0, 1, 2),$$

where we have chosen $\Delta x = \Delta y = 1$ just for simplicity.

We create this 2D mesh using numpy.meshgrid

```
Nx = 3
Ny = 2
Lx = Nx
Ly = Ny
x = np.linspace(0, Lx, Nx+1)
y = np.linspace(0, Ly, Ny+1)
mesh = np.meshgrid(x, y, indexing='ij')
```

At this point `mesh` will actually contain two arrays and we extract them by

```
xij, yij = mesh
```

Now `xij` will contain the x-coordinates of all nodes in the mesh, and `yij` will contain all y-coordinates. Numpy has split up the coordinate pair for all points and put the result into these two arrays. Let us take a look at `xij` and `yij`

```
xij
```

```
array([[0., 0., 0.],
       [1., 1., 1.],
       [2., 2., 2.],
       [3., 3., 3.]])
```

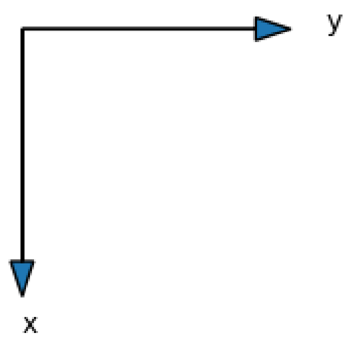
```
yij
```

```
array([[0., 1., 2.],
       [0., 1., 2.],
       [0., 1., 2.],
       [0., 1., 2.]])
```

Hmmm. These arrays seem to be aligned very differently from the mesh that we have been looking at so far. `xij` has shape 4×3 and the x value increases along a column. Likewise, the `yij` array has also shape 4×3, and the y-value increases along a row. So the ordering is as the following arrows show

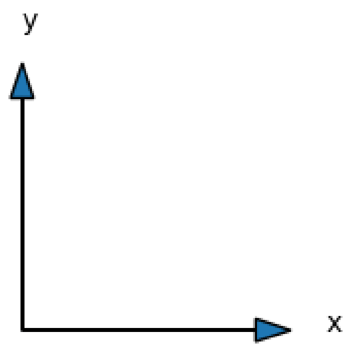
Skip to main content

▶ Show code cell source



This is opposite to a Cartesian mesh that has x varying along the horizontal direction and y varying along the vertical direction:

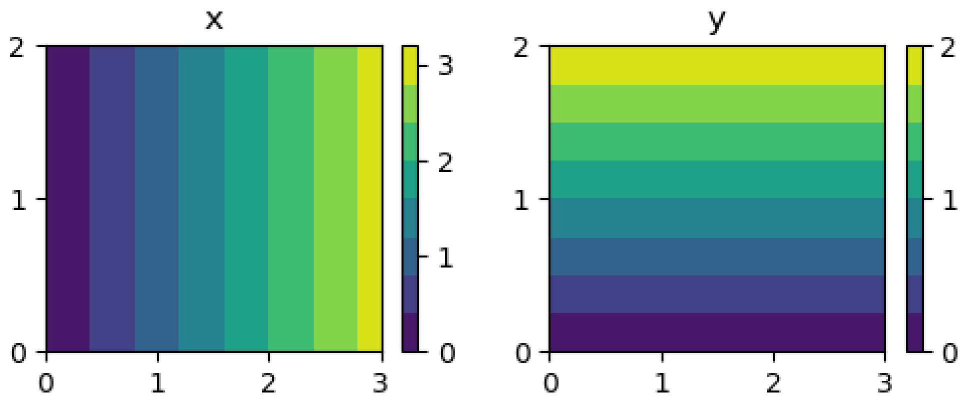
▶ Show code cell source



However, there is no mistake. This is how matrices are stored, with the first index i representing x and the second index j representing y. And if we make contour plots using the meshes `xij` and `yij` it all comes out in a Cartesian plot exactly as expected. Below we create contour plots of both x and y and we notice that x increases along the horizontal axis, whereas y increases in the vertical direction. Just as expected:-)

```
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, sharex=True,
                                figsize=(6, 2))
c0 = ax0.contourf(xij, yij, xij)
c1 = ax1.contourf(xij, yij, yij)
ax0.set_title('x');
ax1.set_title('y');
ax0.set_yticks([0, 1, 2])
ax1.set_yticks([0, 1, 2])
fig.colorbar(c0, ticks=[0, 1, 2, 3]);
fig.colorbar(c1, ticks=[0, 1, 2]);
```

Skip to main content



Sparse meshgrid and broadcasting

The meshgrid creates arrays where all rows are identical for `xij` and all columns for `yij`. This is unnecessary waste of memory, but luckily it is possible to store these 2D arrays using only the memory of one single array. Lets recreate the meshgrid with keyword `sparse=True`:

```
smesh = np.meshgrid(x, y, indexing='ij', sparse=True)
sxij, syij = smesh
sxij, syij
```

```
(array([[0.],
        [1.],
        [2.],
        [3.]]),
 array([[0., 1., 2.]]))
```

Now the meshgrid is sparse and only storing the repeating rows or columns once. However, something has happened to the input $\boldsymbol{x}$ and $\boldsymbol{y}$ arrays, that are originally

```
x, y
```

```
(array([0., 1., 2., 3.]), array([0., 1., 2.]))
```

Note that `x` and `y` are one-dimensional arrays of shape $(4,)$ and $(3,)$. In contrast `sxij` and `syij` are two-dimensional arrays of shape $(4, 1)$ and $(1, 3)$, respectively:

```
print(sxij.shape, syij.shape)
```

Skip to main content

```
(4, 1) (1, 3)
```

The added dimension of length 1 is important for how the array broadcasts. The extra dimension tells Numpy that the array is constant along this other axis. That is, `sxij` is constant along the second axis, no matter what size the second axis is. Similarly, `syij` is constant along the first axis. And since the 2D array is constant along that direction it is not necessary to actually store the numbers. So this is memory efficient.

So what happens if you now have this sparse mesh and you create a mesh function

$$f(x,y) = x + y$$

```
f = sxij + syij
print(f)
```

```
[[0. 1. 2.]
 [1. 2. 3.]
 [2. 3. 4.]
 [3. 4. 5.]]
```

Numpy now sees two arrays of shape $(4,1)$ and $(1,3)$ that needs to be added together. So Numpy will now broadcast both arrays into the only appropriate shape $(4,3)$ and then add them together. The result is exactly the same as if you used `xij` and `yij` directly

```
print(xij+yij)
```

```
[[0. 1. 2.]
 [1. 2. 3.]
 [2. 3. 4.]
 [3. 4. 5.]]
```

Note

Broadcasting is a very useful and memory-saving operation that is highly important to Numpy and the writing of efficient, vectorized, Python code.

Two-dimensional mesh function

Skip to main content

$$u_{ij} = u(x_i, y_j).$$

The mesh function is a dense two-dimensional array of shape $(N_x+1) \times (N_y+1)$ and we will write $U = (u_{ij})_{i,j=0}^{N_x,N_y}$. It may also be considered a matrix.

$$\begin{bmatrix} u_{0,0} & u_{0,1} & \cdots & u_{0,N_y} \\ u_{1,0} & u_{1,1} & \cdots & u_{1,N_y} \\ \vdots & \vdots & \ddots & \vdots \\ u_{N_x,0} & u_{N_x,1} & \cdots & u_{N_x,N_y} \end{bmatrix}$$

We note that the mesh function U is a matrix, like `xij` and `yij`, and in the component u_{ij} i represents row and j represents column.

Also note that sometimes (like above) it makes sense to add a comma between i and j. This has no further significance than to more clearly separate i from j.

Note

Notice that both `xij` and `yij` are mesh functions! They are the coordinates `x` and `y` evaluated on the computational mesh.

Row-major computer storage

In Python (which is written in C) a matrix is row-major. This means that the matrix is stored in memory with a long sequence of numbers row by row like

$$\underbrace{u_{0,0}, u_{0,1}, \cdots, u_{0,N_y}}_{\text{First row}}, \underbrace{u_{1,0}, u_{1,1}, \cdots, u_{1,N_y}}_{\text{Second row}}, \cdots\cdots, \underbrace{u_{N_y,0}, u_{N_y,1}, \cdots, u_{N_y,N_y}}_{\text{Last row}}$$

So the computer memory does not know anything about these numbers belonging to a two-dimensional array. The computer only knows that $(N_x+1)(N_y+1)$ numbers are stored side by side.

We can get the single row i of the U matrix as $u_i = (u_{i,0}, u_{i,1}, \ldots, u_{i,N_y})$. For example, lets create a 3×3 matrix in Python

```
A = np.arange(9).reshape((3, 3))
```

Skip to main content

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

The matrix `A` is laid out in computer memory as

```
A.ravel() # alternative: A.flatten()
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

This is row-major storage. First the first row, then the second row and finally the third. We can easily get individual rows of `A` as, e.g.,

```
A[1]
```

```
array([3, 4, 5])
```

```
A[2]
```

```
array([6, 7, 8])
```

We get the rows easily because the memory is row-major and all the items in a row are side by side in computer memory. However, in order to get a column we need to skip over items. The first column of `A` contains the numbers `[0, 3, 6]`. These numbers are stored in the flattened memory as items 0, 3 and 6. We can look it up as

```
A[:, 0]
```

```
array([0, 3, 6])
```

Finite differences in 2D

Let us move back to the Poisson equation

Skip to main content

$$
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f.
$$

If we now use second order central differences in both directions we get

$$
\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = f_{i,j}.
$$

We have already learned that a second differentiation matrix in one dimension can be written as

$$
D_x^{(2)} = \frac{1}{\Delta x^2} \begin{bmatrix}
2 & -5 & 4 & -1 & 0 & 0 & 0 & 0 \\
1 & -2 & 1 & 0 & 0 & 0 & 0 & \cdots \\
0 & 1 & -2 & 1 & 0 & 0 & 0 & \cdots \\
\vdots & & & \ddots & & & & \cdots \\
\vdots & 0 & 0 & 0 & 1 & -2 & 1 & 0 \\
\vdots & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\
0 & 0 & 0 & 0 & -1 & 4 & -5 & 2
\end{bmatrix}
$$

For the y-direction we will use Δy instead of Δx and we will then write $D_y^{(2)}$.

Note

We will denote the unscaled second derivative matrix $D^{(2)}$. This is the matrix you get with $\Delta x = \Delta y = 1$.

We compute the unscaled matrix $D^{(2)}$ as

```python
import scipy.sparse as sparse

def D2(N):
    D = sparse.diags([1, -2, 1], [-1, 0, 1], (N+1, N+1), 'lil')
    D[0, :4] = 2, -5, 4, -1
    D[-1, -4:] = -1, 4, -5, 2
    return D
print(D2(8).toarray())
```

```
[[ 2. -5.  4. -1.  0.  0.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  1. -2.  1.  0.  0.  0.  0.  0.]
```

Skip to main content

```
 [ 0.  0.  0.  0.  1. -2.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1. -2.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  1. -2.  1.]
 [ 0.  0.  0.  0.  0. -1.  4. -5.  2.]]
```

Now let us consider a very simple function in 2D

$$u(x, y) = x^2,$$

such that

$$\frac{\partial^2 u}{\partial x^2} = 2.$$

Let the domain now be $[0, 4] \times [0, 4]$ and choose $\Delta x = \Delta y = 1$. We create the Cartesian product mesh as follows

```
Nx = 4
Ny = 4
Lx = Nx
Ly = Ny
x = np.linspace(0, Lx, Nx+1)
y = np.linspace(0, Ly, Ny+1)
xij, yij  = np.meshgrid(x, y, indexing='ij')
```

From this mesh we can create the mesh function $u(x, y) = x^2$

```
u = xij**2
print(u)
```

```
[[ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]
 [ 4.  4.  4.  4.  4.]
 [ 9.  9.  9.  9.  9.]
 [16. 16. 16. 16. 16.]]
```

In order to compute $\partial^2 u / \partial x^2$ we can use the one-dimensional derivative matrix $D^{(2)}$ along the first axis of $U = (u_{i,j})_{i,j=0}^{N_x, N_y}$

$$D^{(2)} U = \sum_{k=0}^{N_x} d_{ik}^{(2)} u_{k,j}$$

Skip to main content

So $D^{(2)}U$ is a matrix-matrix product between the two matrices $D^{(2)} \in \mathbb{R}^{(N_x+1)\times(N_x+1)}$ and $U \in \mathbb{R}^{(N_x+1)\times(N_y+1)}$, such that $D^{(2)}U \in \mathbb{R}^{(N_x+1)\times(N_y+1)}$

```
D = D2(Nx)
print(D @ u)
```

```
[[2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]]
```

So we see that the second derivative of u equals 2 in all nodes of the mesh. This means that our code is working:-)

How about the other derivative $\partial^2 u/\partial y^2$? Let us create a mesh function

$$u(x, y) = y^2,$$

such that again the second derivative $\partial^2 u/\partial y^2$ should be 2. Create first u

```
u = yij**2
print(u)
```

```
[[ 0.  1.  4.  9. 16.]
 [ 0.  1.  4.  9. 16.]
 [ 0.  1.  4.  9. 16.]
 [ 0.  1.  4.  9. 16.]
 [ 0.  1.  4.  9. 16.]]
```

Now since the derivative is along the second axis (the y-axis), we cannot simply do $D^{(2)}U$ anymore. If we do we get zero

```
print(D @ u)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Skip to main content

because this is $\partial^2 u/\partial x^2$, which should be zero. To get a derivative along the second axis we need to apply the matrix $D^{(2)}$ along the second axis of U:

$$\sum_{k=0}^{N_y} u_{i,k} d^{(2)}_{j,k}$$

In matrix form this can be written as

$$U (D^{(2)})^T$$

which we can verify as follows

```
print(u @ D.T)
```

```
[[2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]
 [2. 2. 2. 2. 2.]]
```

Again we get the correct result across the entire mesh.

Matrix form of Poisson's equation

Using now that $\partial^2 u/\partial x^2 = D_x^{(2)} U$ and $\partial^2 u/\partial y^2 = U (D_y^{(2)})^T$ we can write Poisson's equation on discretized (and matrix) form as

$$D_x^{(2)} U + U (D_y^{(2)})^T = F, \tag{5}$$

where the mesh function $F = (f(x_i, y_j))_{i,j=0}^{N_x, N_y}$.

However, how can we solve this equation for the unknown U?

The problem now is that we cannot (apparently) write Eq. (5) as we normally do for a linear algebra problem

$$Ax = b, \tag{6}$$

Skip to main content

$$\sum_{j=0}^{N} a_{ij}x_j = b_i, \tag{7}$$

where A is the coefficient matrix, x is the unknown vector and b is the right hand side vector.

However, there is an operation that will help us transform (5) into a regular matrix problem like (6). This operation is called vectorization, or the vec-trick.

Vectorization (the vec-trick)

Consider first for simplicity a 2×2 matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

In general, a row-major vectorization transforms the matrix into a vector as follows

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \longrightarrow \begin{bmatrix} a & b & c & d \end{bmatrix}^T = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

This operation is referred to as `vec(A)` and thus

$$\text{vec}(A) = \begin{bmatrix} a & b & c & d \end{bmatrix}^T$$

Note

Note that we define the output from `vec` as a column vector and not a row vector!

Warning

Notice that it is possible to define a column-major vectorization as well. In that case $\text{vec}(A) = [a\, c\, b\, d]^T$. However, since Python is written in C and C is row-

Skip to main content

Vectorization of our Poisson equation (vectorization applied to both left and right hand side) leads to

$$\text{vec}\left(D_x^{(2)}U + U(D_y^{(2)})^T\right) = \text{vec}(F),$$

and since vectorization is a linear process

$$\text{vec}\left(D_x^{(2)}U\right) + \text{vec}\left(U(D_y^{(2)})^T\right) = \text{vec}(F).$$

Here $\text{vec}(F)$ is simply the ravelling (or flattening) of the matrix as described above. This leaves the left hand side, which contains matrix-matrix products, and these require a special trick for vectorization. But first we will write the equation as

$$\text{vec}\left(D_x^{(2)}UI_y\right) + \text{vec}\left(I_xU(D_y^{(2)})^T\right) = \text{vec}(F),$$

where the diagonal identity matrix $I_i \in \mathbb{R}^{(N_i+1)\times(N_i+1)}$

$$I_i = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ \vdots & & \ddots & & \cdots \\ \vdots & 0 & 0 & 1 & 0 \\ \vdots & 0 & 0 & 0 & 1 \end{bmatrix}$$

is not really doing anything. However, it will help us soon in applying a well known vectorization formula for triple matrix products. It turns out that for any valid triple matrix product of matrices A, B and C, vectorization is defined as

$$\text{vec}(ABC^T) = (A \otimes C)\text{vec}(B),$$

where $A \otimes B$ is the Kronecker product of matrices A and B. We will get back to the Kronecker product soon. Meanwhile we get that

$$\text{vec}(D_x^{(2)}UI_y) = (D_x^{(2)} \otimes I_y)\text{vec}(U),$$

and

Skip to main content

$$\text{vec}(I_x U (D_y^{(2)})^T) = (I_x \otimes D_y^{(2)})\text{vec}(U),$$

and the equation we need to solve is thus

$$\left(D_x^{(2)} \otimes I_y + I_x \otimes D_y^{(2)}\right)\text{vec}(U) = \text{vec}(F).$$

This is a linear equations of type $Ax = b$, where $A = D_x^{(2)} \otimes I_y + I_x \otimes D_y^{(2)}$, $x = \text{vec}(U)$ and $b = \text{vec}(F)$.

Note

Vectorization is a linear process that allows us to transform a matrix into a vector. Through vectorization we can express matrix-multiplication through a larger matrix using the Kronecker product.

The Kronecker product

In general, if A and B are matrices of dimensions $p \times q$ and $r \times s$, respectively, then $A \otimes B$ is the matrix of dimension $pr \times qs$, with $p \times q$ block form, where the i, j block is $a_{ij}B$. The Kronecker product is most simply illustrated for two small matrices of shape 2×2:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \left(\begin{array}{cc|cc} a & b & 2a & 2b \\ c & d & 2c & 2d \\ \hline 3a & 3b & 4a & 4b \\ 3c & 3d & 4c & 4d \end{array}\right)$$

Notice that if one of the matrices is the identity matrix, the picture becomes even simpler

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \left(\begin{array}{cc|cc} a & b & 0 & 0 \\ c & d & 0 & 0 \\ \hline 0 & 0 & a & b \\ 0 & 0 & c & d \end{array}\right)$$

and

Skip to main content

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \left(\begin{array}{cc|cc} a & 0 & b & 0 \\ 0 & a & 0 & b \\ \hline c & 0 & d & 0 \\ 0 & c & 0 & d \end{array}\right)$$

Partial derivatives

Let us try this new theory for partial derivatives. Create first a mesh using meshgrid, then define a function

$$u(x, y) = x^2$$

```
Nx = 4
Ny = 3
Lx = Nx
Ly = Ny
x = np.linspace(0, Lx, Nx+1)
y = np.linspace(0, Ly, Ny+1)
xij, yij = np.meshgrid(x, y, indexing='ij')
U = xij**2
```

Compute the derivative of u using first

$$D^{(2)}U$$

```
D = D2(Nx)
print(D @ U)
```

```
[[2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]]
```

The differentiation works, because the second derivative of x^2 is 2. Now lets use vectorization to compute the same

$$\text{vec}(D^{(2)}U) = (D^{(2)} \otimes I)\text{vec}(U)$$

Skip to main content

```
D2X = sparse.kron(D, sparse.eye(Ny+1))
d2u = D2X @ U.ravel()
print(d2u.reshape(U.shape))
```

```
[[2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]]
```

Works!

How about $\partial^2 u/\partial y^2$? Define first $u = y^2$ such that $\partial^2 u/\partial y^2 = 2$. And compute

$$\text{vec}(U(D^{(2)})^T) = (I \otimes D^{(2)})\text{vec}(U)$$

```
U = yij**2
D2Y = sparse.kron(sparse.eye(Nx+1), D2(Ny))
print(D2Y @ U.ravel())
```

```
[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
```

Works! But the result should be reshaped into a two-dimensional array.

```
print((D2Y @ U.ravel()).reshape(U.shape))
```

```
[[2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]]
```

The Laplace operator

The Laplace operator ∇^2 applied to a two-dimensional field $u(x, y)$ in Cartesian coordinates is

$$\nabla^2 u = \frac{\partial^2 u}{\quad} + \frac{\partial^2 u}{\quad}.$$

Skip to main content

Discretized on a structured Cartesian mesh we get

$$\nabla^2 u = \left(D_x^{(2)} \otimes I_y + I_x \otimes D_y^{(2)}\right) \text{vec}(U).$$

We implement the operator as

```python
def laplace(dx, dy, Nx, Ny):
    D2x = (1./dx**2)*D2(Nx)
    D2y = (1./dy**2)*D2(Ny)
    return (sparse.kron(D2x, sparse.eye(Ny+1)) +
            sparse.kron(sparse.eye(Nx+1), D2y))
```

We are now ready to solve the Poisson equation. Lets use the method of manufactured solutions and guess a solution

$$u(x, y) = x(1-x)y(1-y)\exp(\cos(4\pi x)\sin(2\pi y)),$$

and use a computational domain $\Omega = [0, 1] \times [0, 1]$.

```python
import sympy as sp
x, y = sp.symbols('x,y')
ue = x*(1-x)*y*(1-y)*sp.exp(sp.cos(4*sp.pi*x)*sp.sin(2*sp.pi*y))
```

Compute the right hand side function f

```python
f = ue.diff(x, 2) + ue.diff(y, 2)
```

Create the Cartesian 2D mesh using a function

```python
def mesh2D(Nx, Ny, Lx, Ly):
    x = np.linspace(0, Lx, Nx+1)
    y = np.linspace(0, Ly, Ny+1)
    return np.meshgrid(x, y, indexing='ij')
```

Assemble the problem

```python
Nx = 30
Ny = 30
Lx = 1
Ly = 1
xij, yij = mesh2D(Nx, Ny, Lx, Ly)
```

Skip to main content

```
F = sp.lambdify((x, y), f)(xij, yij)
A = laplace(Lx/Nx, Ly/Ny, Nx, Ny)
```

Boundary conditions

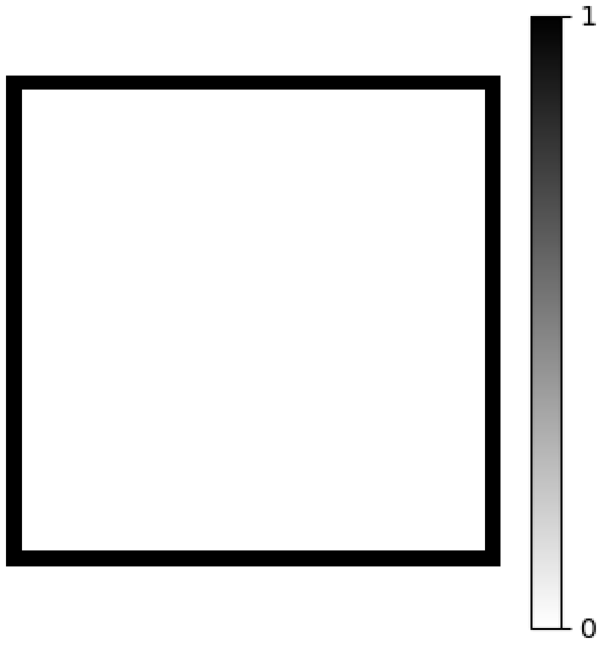
We need to fix the solution at all boundary points. But where are all the boundary points? How do we manipulate A in order to fix Dirichlet boundary conditions?

In order to set Dirichlet boundary conditions we need to set the value of U to zero on all four boundaries:

$$\begin{aligned}
u_{0,j} &= 0 \quad j \in \{0, 1, \ldots, N_y\}, \\
u_{N_x,j} &= 0 \quad j \in \{0, 1, \ldots, N_y\}, \\
u_{i,0} &= 0 \quad i \in \{0, 1, \ldots, N_x\}, \\
u_{i,N_y} &= 0 \quad i \in \{0, 1, \ldots, N_x\}.
\end{aligned}$$

We can do this by identing all rows of `A` corresponding to a boundary point. But which indices in `A` correspond to boundaries? We can find this easily with a little slicing and trickery. Lets create a mesh function `B` that is one on the boundary and zero elsewhere

```
B = np.ones((Nx+1, Ny+1), dtype=bool)
B[1:-1, 1:-1] = 0
fig = plt.figure(figsize=(4, 4))
plt.imshow(B, cmap='gray_r')
plt.gca().axis('off')
plt.colorbar(ticks=[0, 1]);
```



Skip to main content

Now find all the indices in the ravelled (vectorized) array `vec(B)` that equals 1.

```
bnds = np.where(B.ravel() == 1)[0]
print(bnds)
```

```
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  61  62  92  93
 123 124 154 155 185 186 216 217 247 248 278 279 309 310 340 341 371 372
 402 403 433 434 464 465 495 496 526 527 557 558 588 589 619 620 650 651
 681 682 712 713 743 744 774 775 805 806 836 837 867 868 898 899 929 930
 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948
 949 950 951 952 953 954 955 956 957 958 959 960]
```

These are all the indices into `vec(U)` that correspond to a boundary. We need to set the value of U to zero at all these indices. We do that by identing `A` such that each row with an index in `bnds` contains only zeros except the main diagonal which is set to 1.

```
A = A.tolil()
for i in bnds:
    A[i] = 0
    A[i, i] = 1
A = A.tocsr()
```

We also need to modify the right hand side F such that f_{ij} is zero for all points on the boundary. This is easy when `bnds` is available

```
b = F.ravel()
b[bnds] = 0
```

Solve the problem

We have now set up the problem as

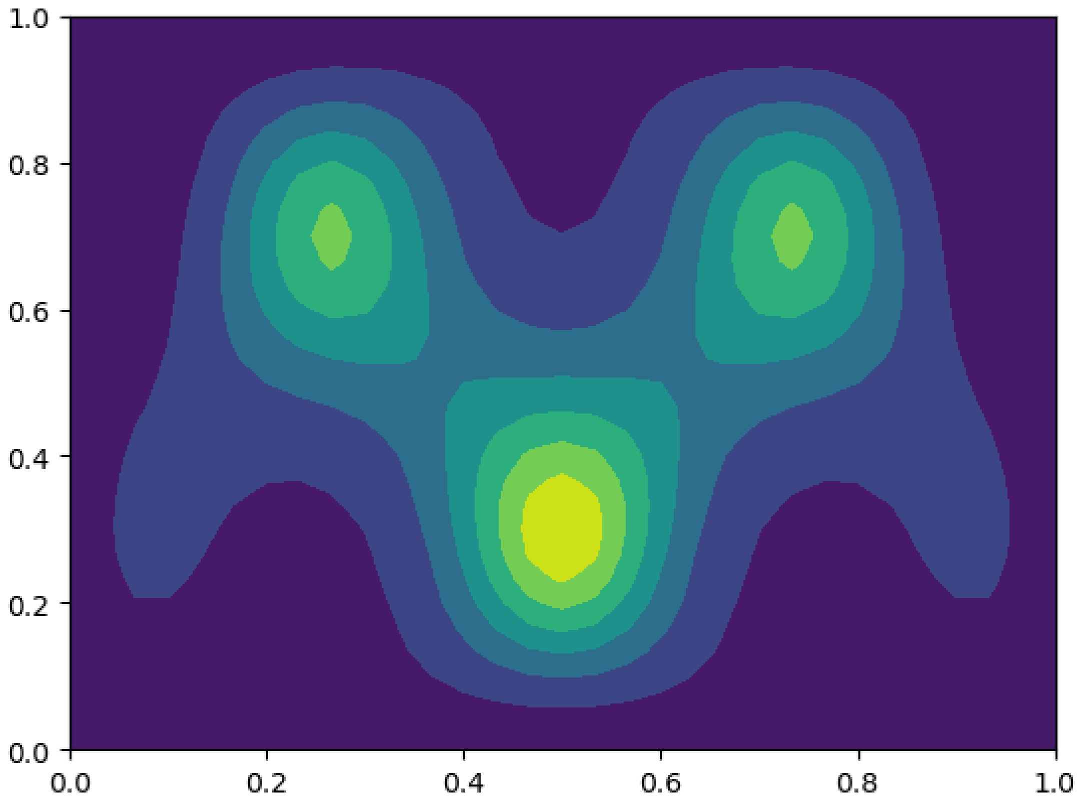
$$Ax = b,$$

and all that is left is to solve the assembled problem using a sparse linear algebra solver

```
U = sparse.linalg.spsolve(A, b)
U = np.reshape(U, (Nx+1, Ny+1))
```

Skip to main content

```
plt.contourf(xij, yij, U);
```



and check that the solution is close to the exact analytical solution

```
dx = Lx/Nx
dy = Ly/Ny
np.sqrt(dx*dy*np.sum((U - sp.lambdify((x, y), ue)(xij, yij))**2))
```

```
0.00038807056403051686
```

Voilà! The solution is correct. You should also verify that the solver works for a variety of spatial discretizations, especially when $N_x \neq N_y$.

Weekly assignments

This week the assignments are to modify poisson.py and poisson2d.py. Study the solver in poisson.py and try to understand how it works.

1. Modify the function test_poisson by adding an appropriate test for the Poisson solver. You may add one or several tests.
2. Implement a 2D Poisson solver in poisson2d.py. Use vectorization, as described in

Skip to main content

such that you can test the solver using the method of manufactured solutions (see the 1D Poisson solver). Implement first the solver for homogeneous Dirichlet boundary conditions. Note that this restricts the possible solutions to all solutions that are zero on the entire boundary.

3. Implement generic Dirichlet boundary conditions for the 2D Poisson solver such that any manufactured solution may be applied.
4. Implement an appropriate test for the solver in the function test_poisson2d.

PS - Remember to add `poisson.py` and `poisson2d.py` to the list of files tested in matmek4270.yml

‹ Previous
Lecture 5

Next ›
Lecture 7