Two-dimensional problems

MATMEK-4270

Prof. Mikael Mortensen, University of Oslo

Important topics todays lecture

- 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 start by considering the Poisson equation

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

which in two-dimensional Cartesian coordinates is

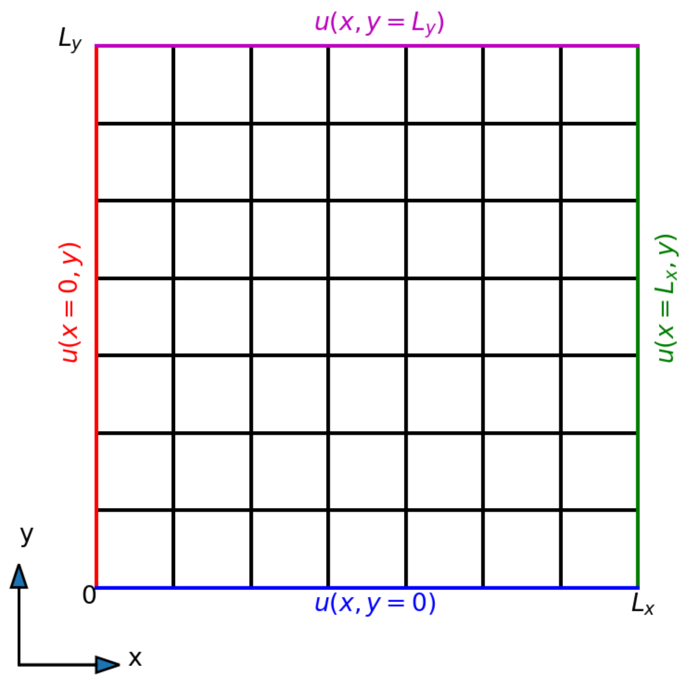
$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f.$$

We consider the equation for a rectangular domain

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

and Dirichlet boundary conditions.

The uniform rectangle



Boundary conditions on any side may be a function of the non-constant variable

$$u(x=0,y) = f(y)$$
$$u(x,y=0) = f(x)$$

where f is a function of one variable, $f : \mathbb{R} \to \mathbb{R}$

Note

The boundary conditions need to be consistent in the corners, because each corner belongs to two sides.

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 $\Delta y = L_y / N_y$. Vector notation for the mesh is

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

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

How do we create the 2D mesh?

Cartesian products!

Consider the two tuples

$$\boldsymbol{u} = (1, 2, 3) \quad \boldsymbol{v} = (4, 5)$$

Compute the **Cartesian product** of these two tuples using the Python package itertools

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

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

The outcome is a tuple containing all possible ordered pairs of items, where the first item is from $\boldsymbol{u}$ and the second from $\boldsymbol{v}$.

Note

- Tuples are *immutable* sequences.

Mathematical description of the Cartesian product

The Cartesian product can be described mathematically as

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

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

Note

- Sets are unordered collections with no duplicate elements.
- Sets are written with curly brackets.
- The pairs (u, v) are not sets since the order of the numbers is important.
- itertools work on any kind of sequence.

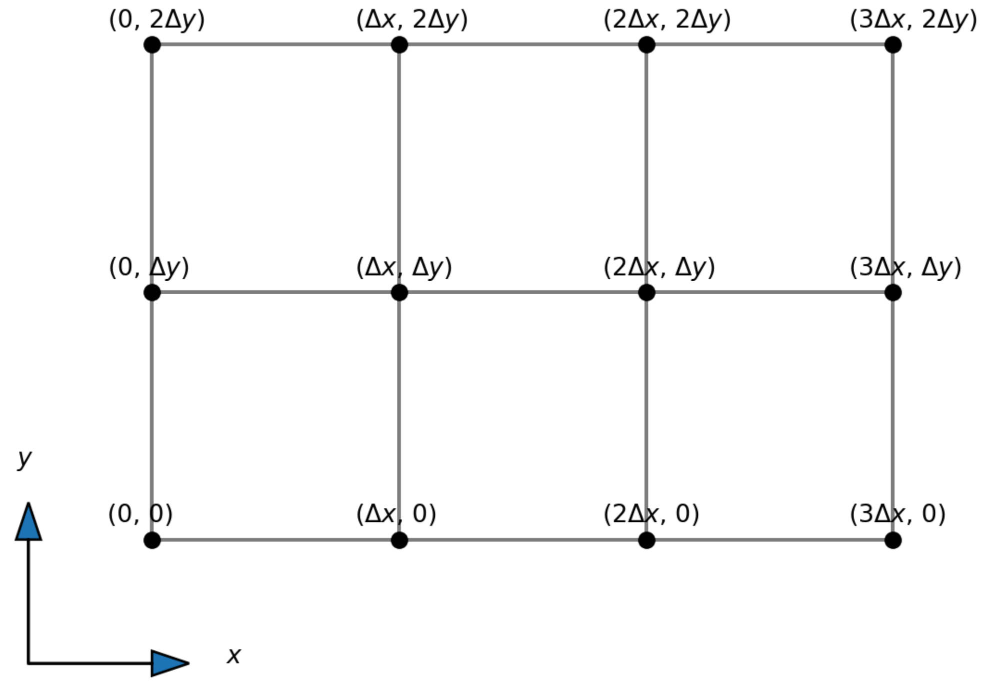
Why is this relevant for computational meshes?

Structured 2D meshes

We have grid lines

$$\boldsymbol{x} = (x_0, x_1, \ldots, x_{N_x}) = (0, \Delta x, 2\Delta x, \ldots, L_x)$$

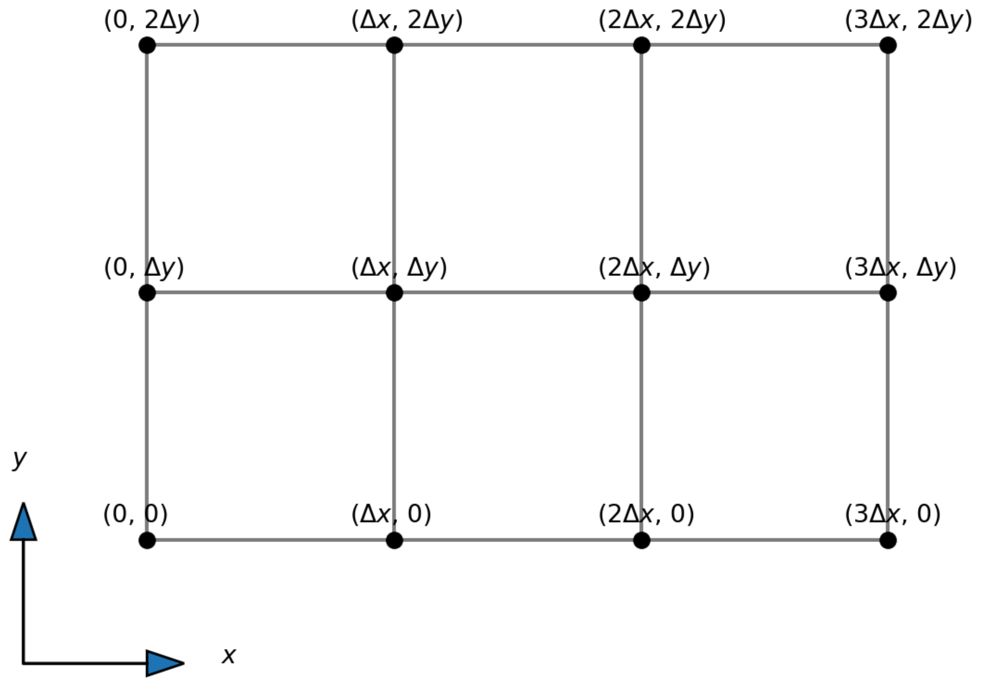
$$\boldsymbol{y} = (y_0, y_1, \ldots, y_{N_y}) = (0, \Delta y, 2\Delta y, \ldots, L_y)$$



The computational mesh is a Cartesian product!

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

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



The Cartesian product is a 1D array of pairs

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

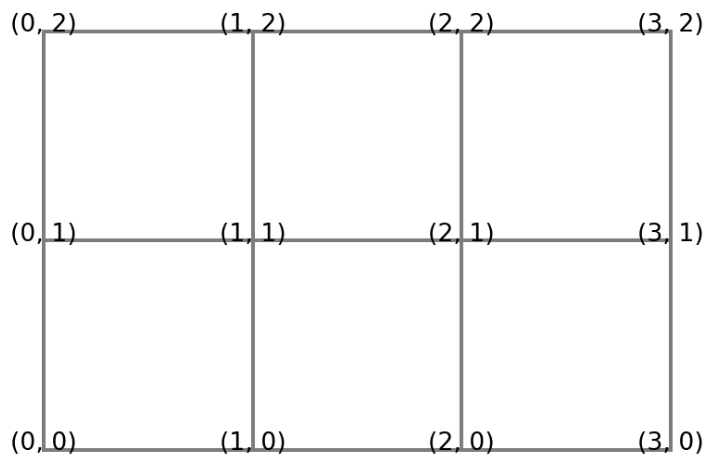
```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2)]
```

It is the same as a double for-loop

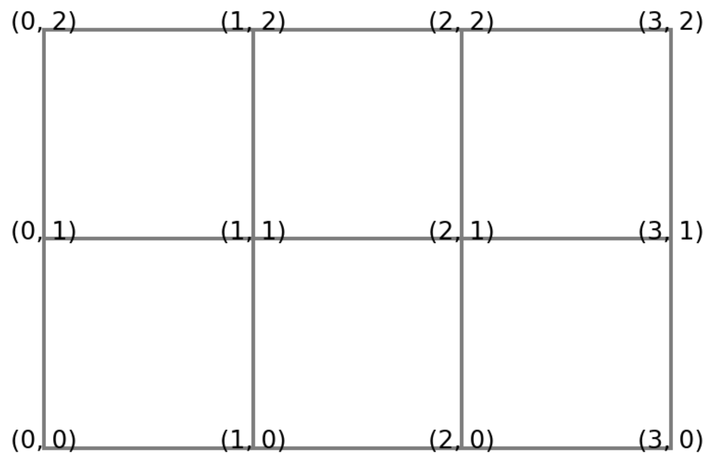
```python
print([(a, b) for a in range(4) for b in range(3)])
```

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

However, the numbers are naturally seen as points in a 2D mesh:



A complicating factor for the mesh



Note

- If this mesh was a 2D 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 second 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!

Numpy meshgrid

Let us now compute our Cartesian mesh using numpy.meshgrid. We have the 1D mesh arrays

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

where we have chosen $\Delta x = \Delta y = 1$ just for simplicity. An implementation goes like

```python
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')
mesh
```

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

Mesh ordering

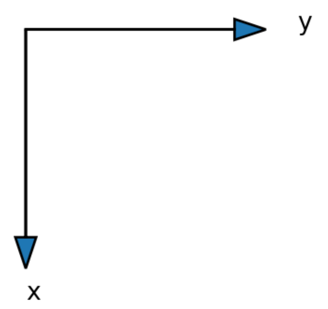
Numpy.meshgrid returns two arrays of shape 4×3

```python
mesh = np.meshgrid(x, y, indexing='ij')
mesh
```

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

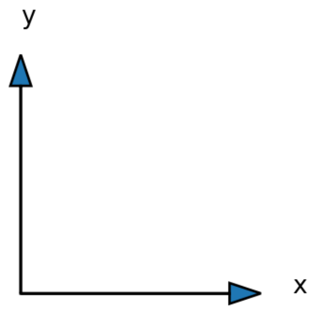
Note

- The first array represents the x-coordinate, and it increases along columns.
- The second array represents the y-coordinate, and it increases along rows.



The mesh is opposite to a Cartesian mesh!

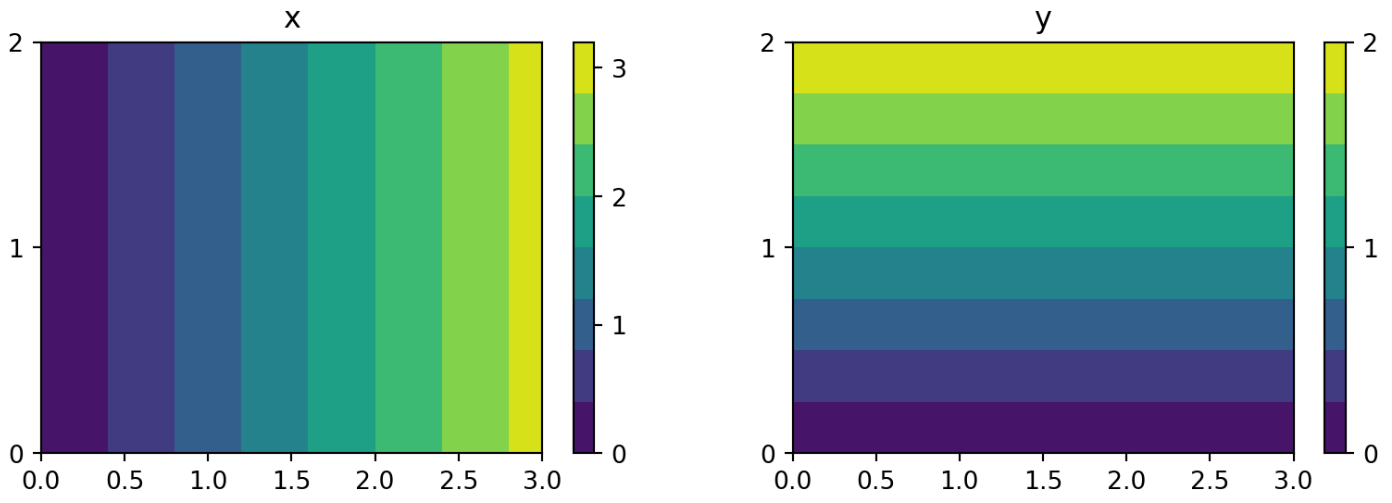
We excpect a coordinate system to be like



However, there is no mistake. This is simply how matrices are stored.

It all comes out nicely in plots though

```python
x, y = mesh
fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, sharex=True,
                               figsize=(10, 3))
c0 = ax0.contourf(x, y, x)
c1 = ax1.contourf(x, y, y)
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]);
```



Just as expected:-)

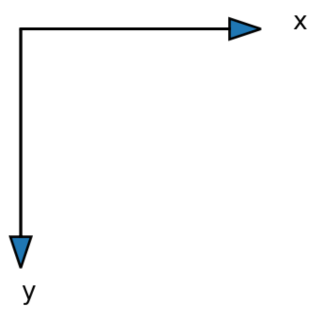
To avoid (or increase?) confusion

You can also choose Cartesian indexing for the meshgrid

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

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

But it is still not completely as expected, because y is increasing downwards!



Sparse meshgrid

It is not necessary to store the complete arrays, because x is not changing along the y-axis and y is not changing along the x-axis.

```
mesh = np.meshgrid(x, y, indexing='ij')
mesh
```

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

Use sparse option:

```
mesh = np.meshgrid(x, y, indexing='ij', sparse=True)
mesh
```

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

The x-array is constant along the y-axis and the y-array is constant along the x-axis

```
print(mesh[0].shape, mesh[1].shape)
```

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

Broadcasting

```python
display(mesh)
display(mesh[0].shape, mesh[1].shape)
```

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

(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, `mesh[0]` is constant along the second axis, no matter what size the second axis is. Similarly, `mesh[1]` 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

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

Broadcasting 2

Compute

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

when x and y are stored using sparse matrices of shape $(4, 1)$ and $(1, 3)$

```
x, y = mesh
print(x.shape, y.shape)
```

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

Now compute $f(x, y)$

```
f = x + y
```

What is the shape of f!

```
f
```

```
array([[0., 1., 2.],
       [1., 2., 3.],
       [2., 3., 4.],
       [3., 4., 5.]])
```

The shape of f is $(4, 3)$! Exactly as it would be without the sparse matrices!

Two-dimensional mesh function

A mesh function on the two-dimensional domain will now be denoted as

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

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.

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

For the component u_{ij}, we note that i represents row and j represents column. This is matrix storage, like we got for meshgrid using `indexing="ij"`.

Row-major computer storage

In Python a matrix is row-major. This means that the matrix that looks like

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

is stored in memory with a long sequence of numbers row by row like

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

Note

The computer 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.

More on row-major storage

We can get the single row i of the U matrix as $u_i = (u_{i,0}, u_{i,1}, \dots, u_{i,N_y})$. For example,

```
U = np.arange(9).reshape((3, 3))
display(U)
display(U[0])
display(U[1])
display(U[2])
```

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

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

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

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

The matrix is laid out in memory as

```
U.ravel()
```

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

To get columns

```
U[:, 0]
```

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

Finite differences in 2D

Let us move back to the Poisson equation

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

Use second order central differences in both directions

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

Note

How to vectorize in 2D? How to use the derivative matrix $D^{(2)}$?

Derivative matrices in 2D

The second order second derivative matrix can be written as

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

We use $D^{(2)} = \Delta x^2 D_x^{(2)}$ for an unscaled matrix.

Hence, for y-direction $D_y^{(2)} = \frac{1}{\Delta y^2} D^{(2)}$.

Compute unscaled matrix using `scipy.sparse`

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

Consider a very simple function in 2D

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

such that

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

Use domain $[0, 4] \times [0, 4]$ and choose $\Delta x = \Delta y = 1$. Create the Cartesian mesh:

```python
Nx, Ny = 4, 4
Lx, Ly = Nx, 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 $u(x, y) = x^2$

```python
U = xij**2
U
```

```
array([[ 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.]])
```

How to compute $\frac{\partial^2 u}{\partial x^2}$?

Use the one-dimensional derivative matrix $D^{(2)}$ along the first axis of $U = (u_{i,j})_{i,j=0}^{N_x,N_y}$

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

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 $D^{(2)}U \in \mathbb{R}^{(N_x+1)\times(N_y+1)}$

```python
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.]]
```

Voila! The second derivative is 2 for all mesh points!

How about the other derivative $\partial^2 u/\partial y^2$?

Create another 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

```python
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

```python
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.]]
```

Derivatives along y-direction

To get a derivative along the second axis we need to apply the matrix $D^{(2)}$ along the second axis of U:

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

In matrix form this can be written as

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

which we can verify as follows

```python
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.]]
```

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

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

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

ⓘ How can we solve this equation for the unknown matrix U?

Can we write this equation as

$$Ax = b$$

where A is the coefficient matrix, x is the unknown vector and b is the rhs vector?

ⓘ Can we transform our unknown matrix U into a vector?

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

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

Note

The vec-trick transforms a matrix into a vector! Just like numpy.ravel.

What about matrix products?

Poisson’s equation contains matrix matrix products

$$D_x^{(2)}U + U(D_y^{(2)})$$

How can these be vectorized if we want to ravel U?

Enter the Kronecker product!

Vectorization of matrix-matrix products

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

Let

$$C = AB \quad \text{such that} \quad c_{ij} = \sum_{k \in (0,1)} a_{ik} b_{kj}$$

And as before

$$\text{vec}(C) = \begin{bmatrix} c_{00} & c_{01} & c_{10} & c_{11} \end{bmatrix}^T \quad \text{vec}(B) = \begin{bmatrix} b_{00} & b_{01} & b_{10} & b_{11} \end{bmatrix}^T$$

How do we find the matrix M:

$$\text{vec}(C) = \text{vec}(AB) = M\text{vec}(B)$$

How to vectorize matrix-matrix products ?

$$\mathrm{vec}(C) = \mathrm{vec}(AB) = M\mathrm{vec}(B)$$

$$\underbrace{\begin{bmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{bmatrix}}_{\mathrm{vec}(C)} = \underbrace{\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} b_{00} \\ b_{01} \\ b_{10} \\ b_{11} \end{bmatrix}}_{\mathrm{vec}(B)}$$

How to find M?

Note that from $c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j}$ we have:

$$\begin{aligned} c_{00} &= a_{00}b_{00} + a_{01}b_{10} \\ c_{01} &= a_{00}b_{01} + a_{01}b_{11} \\ c_{10} &= a_{10}b_{00} + a_{11}b_{10} \\ c_{11} &= a_{10}b_{01} + a_{11}b_{11} \end{aligned}$$

Hence

$$C = AB \quad \text{and thus} \quad c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j}$$

is obtained for the following matrix M

$$\underbrace{\begin{bmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{bmatrix}}_{\mathrm{vec}(C)} = \underbrace{\begin{bmatrix} a_{00} & 0 & a_{01} & 0 \\ 0 & a_{00} & 0 & a_{01} \\ a_{10} & 0 & a_{11} & 0 \\ 0 & a_{10} & 0 & a_{11} \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} b_{00} \\ b_{01} \\ b_{10} \\ b_{11} \end{bmatrix}}_{\mathrm{vec}(B)}$$

Just check that this corresponds to

$$\begin{aligned} c_{00} &= a_{00}b_{00} + a_{01}b_{10} \\ c_{01} &= a_{00}b_{01} + a_{01}b_{11} \\ c_{10} &= a_{10}b_{00} + a_{11}b_{10} \\ c_{11} &= a_{10}b_{01} + a_{11}b_{11} \end{aligned}$$

The Kronecker product

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:

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

For diagonal matrices it is 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

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

The vec-trick for matrix products

A very useful result is

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

Hence in the example from before with $C = AB$:

$$\mathrm{vec}(C) = \mathrm{vec}(AB) = \mathrm{vec}(ABI) = (A \otimes I)\mathrm{vec}(B)$$

$$A \otimes I = \begin{bmatrix} a_{00} & 0 & a_{01} & 0 \\ 0 & a_{00} & 0 & a_{01} \\ a_{10} & 0 & a_{11} & 0 \\ 0 & a_{10} & 0 & a_{11} \end{bmatrix}$$

where I is the identity matrix.

The vec-trick for derivatives in 2D

Consider the second derivative in the x-direction

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

With regular matrices we can compute this as

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

```python
U = xij**2
D @ U
```

```
array([[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.]])
```

How to compute the second derivative using vectorization?

Second derivative with vectorization

$$\mathrm{vec}(D_x^{(2)}U) = \mathrm{vec}(D_x^{(2)}UI) = \underbrace{(D_x^{(2)} \otimes I)}_{M}\mathrm{vec}(U)$$

Implementation using `scipy.sparse`

```python
M = sparse.kron(D, sparse.eye(Ny+1))
d2u = M @ 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.]
 [2. 2. 2. 2. 2.]]
```

Same exact result! Since the result `d2u` is a 1D array, we reshape into matrix form.

Vectorization can be applied to equations

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

and since vectorization is a linear process

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

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

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

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 Laplace operator

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

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

Discretized on a structured Cartesian mesh we get

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

Implementation Laplace

```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 almost ready to solve the Poisson equation

$$\nabla^2 u(x,y) = f(x,y), \quad (x,y) \in \Omega = (0,1) \times (0,1)$$

with Dirichlet boundary conditions $u(x,y) = 0$ at all four sides.

ⓘ But how to fix boundary conditions?

Boundary conditions

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

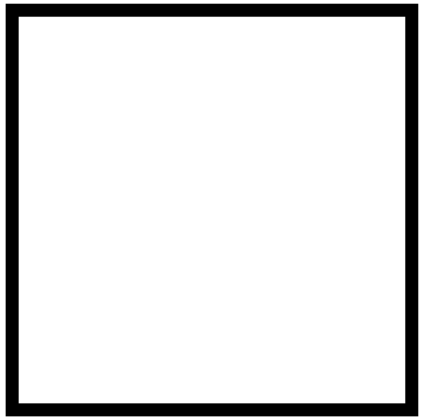
$$\begin{aligned} u_{0,j} &= 0 \quad j \in \{0, 1, \ldots, N_y\}, \\ u_{N_x,j} &= 0 \quad j \in \{0, 1, \ldots, N_y\}, \\ u_{i,0} &= 0 \quad i \in \{0, 1, \ldots, N_x\}, \\ u_{i,N_y} &= 0 \quad i \in \{0, 1, \ldots, N_x\}. \end{aligned}$$

But where are all the boundary points? Can we manipulate the matrix M in order to fix Dirichlet boundary conditions?

We can do this by identing all rows of `M` corresponding to a boundary point. But which indices in `M` 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

Find the boundary points

```python
Nx, Ny = 30, 30
B = np.ones((Nx+1, Ny+1), dtype=bool)
B[1:-1, 1:-1] = 0
fig = plt.figure(figsize=(3, 3))
plt.imshow(B, cmap='gray_r')
plt.gca().axis('off')
```



Find all the points in the vectorized array $\mathrm{vec}(B)$ that equals 1:

```python
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
```

Ident vectorized Laplace matrix for all boundary points

```python
Lx, Ly = 1, 1
A = laplace(Lx/Nx, Ly/Ny, Nx, Ny)
A = A.tolil()
for i in bnds:
    A[i] = 0
    A[i, i] = 1
A = A.tocsr()
```

Solve problem

Lets use the method of manufactured solutions and guess a solution

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

The right hand side function $f(x, y)$ is then

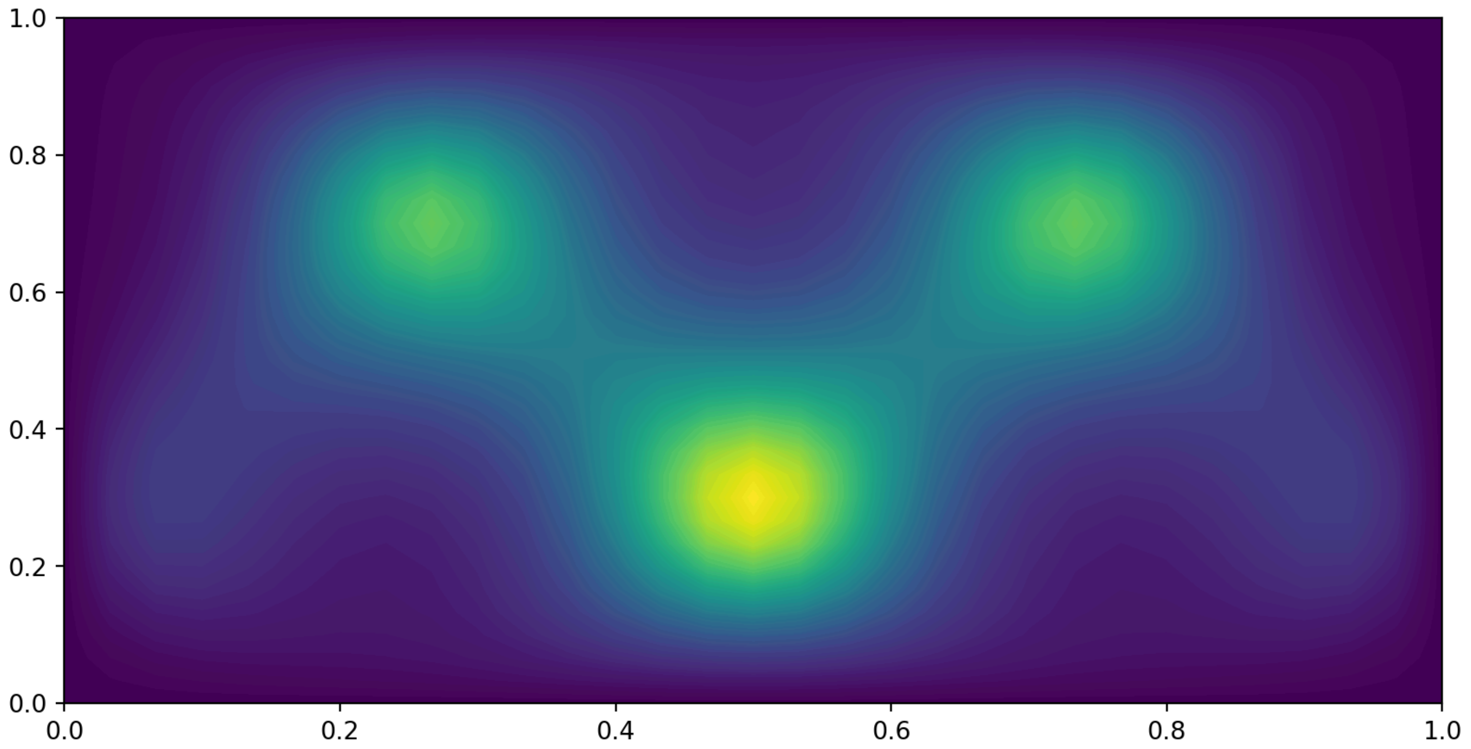
```python
import sympy as sp
x, y = sp.symbols('x,y')
ue = x*(1-x)*y*(1-y)*sp.exp(sp.cos(4*sp.pi*x)*sp.sin(2*sp.pi*y))
f = ue.diff(x, 2) + ue.diff(y, 2)
```

The 2D mesh and mesh function F can be computed as

```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')
xij, yij = mesh2D(Nx, Ny, Lx, Ly)
F = sp.lambdify((x, y), f)(xij, yij)
b = F.ravel()
b[bnds] = 0
```

Solve!

```python
U = sparse.linalg.spsolve(A, b)
U = np.reshape(U, (Nx+1, Ny+1))
plt.contourf(xij, yij, U, 100);
```



Summary

- Two-dimensional problems on simple Cartesian domains make use of Cartesian products to create structured uniform meshes.
- In 2D the unknown meshfunction $U = (u_{ij})_{i,j=0}^{N_x,N_y}$ is a matrix and not a vector.
- In order to use regular linear algebra solvers and recast the problem as

$$Ax = b$$

  we make use of the veck-trick and the Kronecker product

- The vec-trick transforms a matrix into a vector

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

- The vec-trick transforms matrix-matrix products using the Kronecker product

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

Summary continued

- The Poisson equation in 2D reads

$$D_x^{(2)} U + U (D_y^{(2)})^T = F$$

- Regular linear algebra cannot easily solve the above since the unknown is a matrix. Hence we transform the equation using the vec-trick

$$\underbrace{\left(D_x^{(2)} \otimes I_y + I_x \otimes D_y^{(2)}\right)}_{M} \mathrm{vec}(U) = \mathrm{vec}(F)$$

- and solve

$$\mathrm{vec}(U) = M^{-1} \mathrm{vec}(F)$$