# 1 Diffusion in 2D

We now address a diffusion in two space dimensions:

$$\frac{\partial u}{\partial t} = \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} \right) + f(x, y), \tag{1}$$

in a domain

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

with $u = 0$ on the boundary and $u(x, y, 0) = I(x, y)$ as initial condition.

## 1.1 Discretization

For generality, it is natural to use a $\theta$-rule for the time discretization. Standard, second-order accurate finite differences are used for the spatial derivatives. We sample the PDE at a space-time point $(i, j, n + \frac{1}{2})$ and apply the difference approximations:

$$\begin{aligned}[D_t u]^{n+\frac{1}{2}} &= \theta[\alpha(D_x D_x u + D_y D_y u) + f]^{n+1} + \\ &\quad (1-\theta)[\alpha(D_x D_x u + D_y D_y u) + f]^{n}\,.\end{aligned} \tag{2}$$

Written out,

$$\begin{aligned}\frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} &= \theta(\alpha(\frac{u^{n+1}_{i-1,j} - 2^{n+1}_{i,j} + u^{n+1}_{i+1,j}}{\Delta x^2}) + (\frac{u^{n+1}_{i,j-1} - 2^{n+1}_{i,j} + u^{n+1}_{i,j+1}}{\Delta y^2})) + f^{n+1}_{i,j}) + \\ &\quad (1-\theta)(\alpha(\frac{u^{n}_{i-1,j} - 2^{n}_{i,j} + u^{n}_{i+1,j}}{\Delta x^2}) + (\frac{u^{n}_{i,j-1} - 2^{n}_{i,j} + u^{n}_{i,j+1}}{\Delta y^2})) + + f^{n}_{i,j})\end{aligned} \tag{3}$$

We collect the unknowns on the left-hand side

$$\begin{aligned}&u^{n+1}_{i,j} - \theta\left(F_x(u^{n+1}_{i-1,j} - 2^{n+1}_{i,j} + u^{n+1}_{i,j}) + F_y(u^{n+1}_{i,j-1} - 2^{n+1}_{i,j} + u^{n+1}_{i,j+1})\right) = u^n_{i,j} + \\ &\quad (1-\theta)\left(F_x(u^{n}_{i-1,j} - 2^{n}_{i,j} + u^{n}_{i,j}) + F_y(u^{n}_{i,j-1} - 2^{n}_{i,j} + u^{n}_{i,j+1})\right) + \\ &\quad \theta\Delta t f^{n+1}_{i,j} + (1-\theta)\Delta t f^n_{i,j},\end{aligned} \tag{4}$$

where

$$F_x = \frac{\alpha \Delta t}{\Delta x^2}, \quad F_y = \frac{\alpha \Delta t}{\Delta x^y},$$

are the Fourier numbers in $x$ and $y$ direction, respectively.

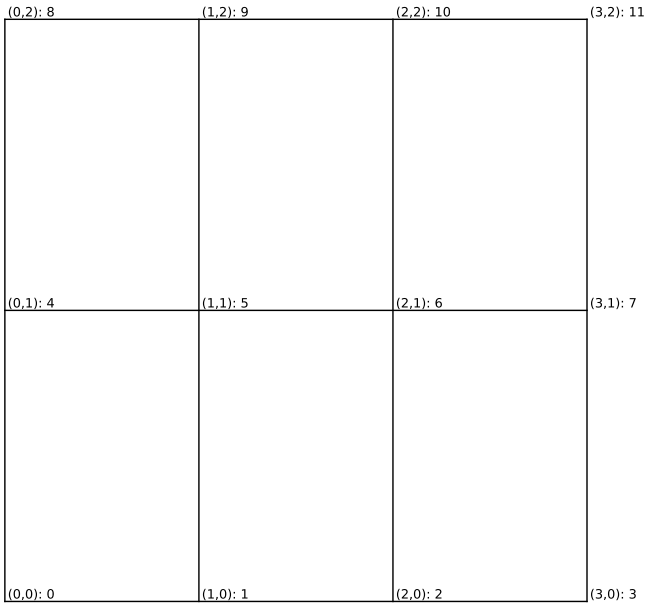



Figure 1: 3x2 2D mesh.

## 1.2 Numbering of mesh points versus equations and unknowns

The equations (4) are coupled at the new time level $n+1$. That is, we must solve a system of (linear) algebraic equations, which we will write as $Ac = b$, where $A$ is the coefficient matrix, $c$ is the vector of unknowns, and $b$ is the right-hand side.

Let us examine the equations in $Ac = b$ on a mesh with $N_x = 3$ and $N_y = 2$ cells in each direction. The spatial mesh is depicted in Figure 1. The equations at the boundary just implement the boundary condition $u = 0$:

$$u_{0,0}^{n+1} = u_{1,0}^{n+1} = u_{2,0}^{n+1} = u_{3,0}^{n+1} = u_{0,1}^{n+1} = u_{3,1}^{n+1} = u_{0,2}^{n+1} = u_{1,2}^{n+1} = u_{2,2}^{n+1} = u_{3,2}^{n+1} = 0\,.$$

We are left with two interior points, with $i = 1$, $j = 1$ and $i = 2$, $j = 1$. The corresponding equations are

$$\begin{aligned} u_{i,j}^{n+1} - \theta \left(F_x(u_{i-1,j}^{n+1} - 2_{i,j}^{n+1} + u_{i,j}^{n+1}) + F_y(u_{i,j-1}^{n+1} - 2_{i,j}^{n+1} + u_{i,j+1}^{n+1})\right) = u_{i,j}^{n} + \\ (1-\theta) \left(F_x(u_{i-1,j}^{n} - 2_{i,j}^{n} + u_{i,j}^{n}) + F_y(u_{i,j-1}^{n} - 2_{i,j}^{n} + u_{i,j+1}^{n})\right) + \\ \theta \Delta t f_{i,j}^{n+1} + (1-\theta)\Delta t f_{i,j}^{n}, \end{aligned}$$

There are in total 12 unknowns $u_{i,j}^{n+1}$ for $i = 0, 1, 2, 3$ and $j = 0, 1, 2$. To solve the equations, we need to form a matrix system $Ac = b$. In that system,

the solution vector $c$ can only one index. Thus, we need a numbering of the unknowns with one index, not two as used in the mesh. We introduce a mapping $m(i,j)$ from a mesh point with indices $(i,j)$ to the corresponding unknown $p$ in the equation system:

$$p = m(i,j) = j(N_x+1) + i\,.$$

When $i$ and $j$ runs through their values we see the following mapping to $p$:

$$\begin{aligned}
&(0,0) \to 0,\ (0,1) \to 1,\ (0,2) \to 2,\ (0,3) \to 3,\\
&(1,0) \to 4,\ (1,1) \to 5,\ (1,2) \to 6,\ (1,3) \to 7,\\
&(2,0) \to 8,\ (2,1) \to 9,\ (2,2) \to 10,\ (2,3) \to 11\,.
\end{aligned}$$

That is, we number the points along the $x$ axis, starting with $y = 0$, and the progress one horizontal mesh line at a time. In Figure 1 you can see that the $(i,j)$ and the corresponding single index $(p)$ are listed for each mesh point.

We could equally well numbered the equations in other ways, e.g., let the $j$ index be the fastest varying index: $p = m(i,j) = i(N_y+1) + j$.

Let us form the coefficient matrix $A$, or more precisely, insert matrix element (according Python's convention with zero as base index) for each of the nonzero elements in $A$ (the indices run through the values of $p$, i.e., $p = 0, \ldots, 11$):

$$\begin{pmatrix}
(0,0) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & (1,1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & (2,2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & (3,3) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & (4,4) & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & (5,1) & 0 & 0 & (5,4) & (5,5) & (5,6) & 0 & 0 & (5,9) & 0 & 0\\
0 & 0 & (6,2) & 0 & 0 & (6,5) & (6,6) & (6,7) & 0 & 0 & (6,10) & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & (7,7) & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (8,8) & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (9,9) & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (10,10) & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & (11,11)
\end{pmatrix}$$

Here is a more compact visualization of the coefficient matrix where we insert dots for zeros and bullets for non-zero elements:

$$\begin{pmatrix}
\bullet & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & \bullet & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & \cdot & \bullet & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & \cdot & \cdot & \bullet & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \bullet & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot\\
\cdot & \bullet & \cdot & \cdot & \bullet & \bullet & \bullet & \cdot & \cdot & \bullet & \cdot & \cdot\\
\cdot & \cdot & \bullet & \cdot & \cdot & \bullet & \bullet & \bullet & \cdot & \cdot & \bullet & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \bullet & \cdot & \cdot & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \bullet & \cdot & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \bullet & \cdot & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \bullet & \cdot\\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \bullet
\end{pmatrix}$$

It is clearly seen that most of the elements are zero. This is a general feature of coefficient matrices arising from discretizing PDEs by finite difference methods. We say that the matrix is *sparse*.

Let $A_{p,q}$ be the value of element $(p,q)$ in the coefficient matrix $A$, where $p$ and $q$ now correspond to the numbering of the unknowns in the equation system. We have $A_{p,q} = 1$ for $p = q = 0, 1, 2, 3, 4, 7, 8, 9, 10, 11$, corresponding to all the known boundary values. Let $p$ be $m(i,j)$, i.e., the single index corresponding to mesh point $(i,j)$. Then we have

$$A_{m(i,j),m(i,j)} = A_{p,p} = 1 + \theta(F_x + F_y), \tag{5}$$
$$A_{p,m(i-1,j)} = A_{p,p-1} = -\theta F_x, \tag{6}$$
$$A_{p,m(i+1,j)} = A_{p,p+1} = -\theta F_x, \tag{7}$$
$$A_{p,m(i,j-1)} = A_{p,p-(N_x+1)} = -\theta F_y, \tag{8}$$
$$A_{p,m(i,j+1)} = A_{p,p+(N_x+1)} = -\theta F_y, \tag{9}$$
$$\tag{10}$$

for the equations associated with the two interior mesh points. At these interior points, the single index $p$ takes on the specific values $p = 5, 6$, corresponding to the values $(1,1)$ and $(1,2)$ of the pair $(i,j)$.

The above values for $A_{p,q}$ can be inserted in the matrix:

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\theta F_y & 0 & 0 & -\theta F_x & 1+2\theta F_x & -\theta F_x & 0 & 0 & -\theta F_y & 0 & 0 \\
0 & 0 & -\theta F_y & 0 & 0 & -\theta F_x & 1+2\theta F_x & -\theta F_x & 0 & 0 & -\theta F_y & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

The corresponding right-hand side vector in the equation system has the entries $b_p$, where $p$ numbers the equations. We have

$$b_0 = b_1 = b_2 = b_3 = b_4 = b_7 = b_8 = b_9 = b_{10} = b_{11} = 0,$$

for the boundary values. For the equations associated with the interior points, we get for $p = 5, 6$, corresponding to $i = 1, 2$ and $j = 1$:

$$b_p = u_i + (1-\theta)\left(F_x(u^n_{i-1,j} - 2^n_{i,j} + u^n_{i,j}) + F_y(u^n_{i,j-1} - 2^n_{i,j} + u^n_{i,j+1})\right) + \theta\Delta t f^{n+1}_{i,j} + (1-\theta)\Delta t f^n_{i,j}\,.$$

Recall that $p = m(i,j) = j(N_x+1) + j$ in this expression.

We can, as an alternative, leave the boundary mesh points out of the matrix system. For a mesh with $N_x = 3$ and $N_y = 2$ there are only two internal mesh points whose unknowns will enter the matrix system. We must now number the unknowns at the interior points:

$$p = (j-1)(N_x - 1) + i,$$

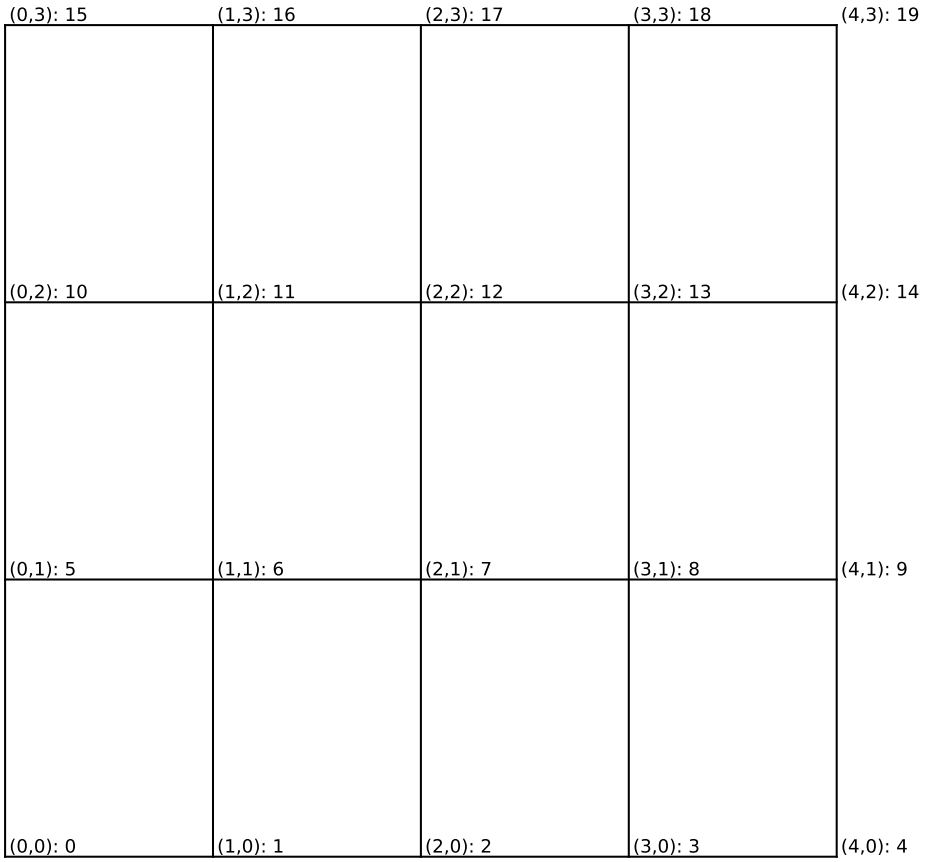
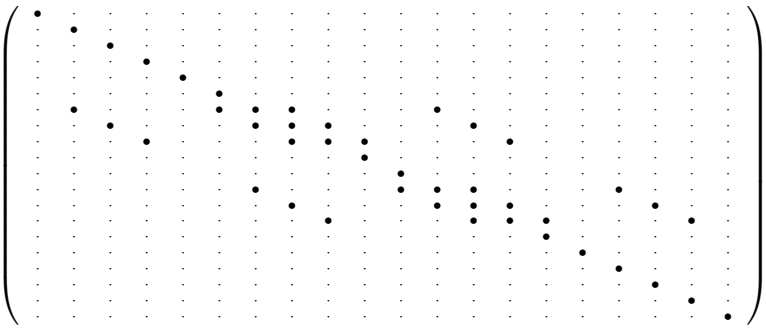



Figure 2: 4x3 2D mesh.

for $i = 1, \ldots, N_x - 1$, $j = 1, \ldots, N_y - 1$.

**hpl 1**: Fill in details.

We can continue with illustrating a bit larger mesh, $N_x = 4$ and $N_y = 3$, see Figure 2. The corresponding coefficient matrix with dots for zeros and bullets for non-zeroes look as follows (values at boundary points are included in the equation system):



**The coefficient matrix is banded.**

Besides being sparse, we observe that the coefficient matrix is *banded*: it has five distinct bands. We have the diagonal $A_{i,i}$, the subdiagonal $A_{i-1,j}$, the superdiagonal $A_{i,i+1}$, a lower diagonal $A_{i,i-(Nx+1)}$, and an upper diagonal $A_{i,i+(Nx+1)}$. These diagonals are important for utilizing efficient algorithms for solving the equation system.

## 1.3 Algorithm for setting up the coefficient matrix

We looked at a specific mesh in the previous section, formulated the equations, and saw what the corresponding coefficient matrix and right-hand side are. Now our aim is to set up a general algorithm, for any choice of $N_x$ and $N_y$, that produces the coefficient matrix and the right-hand side vector. We start with a zero matrix and vector, run through each mesh point, and fill in the values depending on whether the mesh point is an interior point or on the boundary.

- for $i = 0, \ldots, N_x$
  - for $j = 0, \ldots, N_y$
    - $p = j(N_x + 1) + i$
    - if point $(i, j)$ is on the boundary:
      - $A_{p,p} = 1$, $b_p = 0$
    - else:
      - fill $A_{p,m(i-1,j)}$, $A_{p,m(i+1,j)}$, $A_{p,m(i,j)}$, $A_{p,m(i,j-1)}$, $A_{p,m(i,j+1)}$, and $b_p$

To ease the test on whether $(i, j)$ is on the boundary or not, we can split the loops a bit, starting with the boundary line $j = 0$, then treat the interior lines $1 \leq j < N_y$, and finally treat the boundary line $j = N_y$:

- for $i = 0, \ldots, N_x$
  - boundary $j = 0$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$
- for $j = 0, \ldots, N_y$
  - boundary $i = 0$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$
  - for $i = 1, \ldots, N_x - 1$
    - interior point $p = j(N_x + 1) + i$
    - fill $A_{p,m(i-1,j)}$, $A_{p,m(i+1,j)}$, $A_{p,m(i,j)}$, $A_{p,m(i,j-1)}$, $A_{p,m(i,j+1)}$, and $b_p$
  - boundary $i = N_x$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$
- for $i = 0, \ldots, N_x$
  - boundary $j = N_y$: $p = j(N_x + 1) + i$, $A_{p,p} = 1$

The right-hand side is set up as follows.

- for $i = 0, \ldots, N_x$
  - boundary $j = 0$: $p = j(N_x + 1) + i$, $b_p = 0$
- for $j = 0, \ldots, N_y$
  - boundary $i = 0$: $p = j(N_x + 1) + i$, $b_p = 0$
  - for $i = 1, \ldots, N_x - 1$
    - interior point $p = j(N_x + 1) + i$
    - fill $b_p$
  - boundary $i = N_x$: $p = j(N_x + 1) + i$, $b_p = 0$
- for $i = 0, \ldots, N_x$
  - boundary $j = N_y$: $p = j(N_x + 1) + i$, $b_p = 0$

## 1.4 Implementation with a dense coefficient matrix

The goal now is to map the algorithms in the previous section to Python code. One should for computational efficiency reasons take advantage of the fact that the coefficient matrix is sparse and/or banded, i.e., take advantage of all the zeros; however, we first demonstrate how to fill an $N \times N$ dense square matrix, where $N$ is the number of unknowns, here $N = (N_x + 1)(N_y + 1)$. The dense matrix is much easier to understand than the sparse matrix case.

```
import numpy as np

def solver_dense(
    I, a, f, Lx, Ly, Nx, Ny, dt, T, theta=0.5, user_action=None):
    """
    Solve u_t = a*(u_xx + u_yy) + f, u(x,y,0)=I(x,y), with u=0
    on the boundary, on [0,Lx]x[0,Ly]x[0,T], with time step dt,
    using the theta-scheme.
    """
    x = np.linspace(0, Lx, Nx+1)       # mesh points in x dir
    y = np.linspace(0, Ly, Ny+1)       # mesh points in y dir
    dx = x[1] - x[0]
    dy = y[1] - y[0]

    dt = float(dt)                     # avoid integer division
    Nt = int(round(T/float(dt)))
    t = np.linspace(0, Nt*dt, Nt+1)   # mesh points in time

    # Mesh Fourier numbers in each direction
    Fx = a*dt/dx**2
    Fy = a*dt/dy**2
```

The $u_{i,j}^{n+1}$ and $u_{i,j}^{n}$ mesh functions are represented by their spatial values at the mesh points:

```
u   = np.zeros((Nx+1, Ny+1))      # unknown u at new time level
u_1 = np.zeros((Nx+1, Ny+1))      # u at the previous time level
```

It is a good habit (for extensions) to introduce index sets for all mesh points:

```
Ix = range(0, Nx+1)
Iy = range(0, Ny+1)
It = range(0, Nt+1)
```

The initial condition is easy to fill in:

```
# Load initial condition into u_1
for i in Ix:
    for j in Iy:
        u_1[i,j] = I(x[i], y[j])
```

The memory for the coefficient matrix and right-hand side vector is allocated by

```
N = (Nx+1)*(Ny+1)  # no of unknowns
A = np.zeros((N, N))
b = np.zeros(N)
```

The filling of `A` goes like this:

```
m = lambda i, j: j*(Nx+1) + i

# Equations corresponding to j=0, i=0,1,... (u known)
j = 0
for i in Ix:
    p = m(i,j);  A[p, p] = 1

# Loop over all internal mesh points in y diretion
# and all mesh points in x direction
for j in Iy[1:-1]:
    i = 0;  p = m(i,j);  A[p, p] = 1   # Boundary
```

```
    for i in Ix[1:-1]:                # Interior points
        p = m(i,j)
        A[p, m(i,j-1)] = - theta*Fy
        A[p, m(i-1,j)] = - theta*Fx
        A[p, p]        = 1 + 2*theta*(Fx+Fy)
        A[p, m(i+1,j)] = - theta*Fx
        A[p, m(i,j+1)] = - theta*Fy
    i = Nx;  p = m(i,j);  A[p, p] = 1  # Boundary
# Equations corresponding to j=Ny, i=0,1,... (u known)
j = Ny
for i in Ix:
    p = m(i,j);  A[p, p] = 1
```

Since `A` is independent of time, it can be filled once and for all before the time loop. The right-hand side vector must be filled at each time level inside the time loop:

```
import scipy.linalg

for n in It[0:-1]:
    # Compute b
    j = 0
    for i in Ix:
        p = m(i,j);  b[p] = 0           # Boundary
    for j in Iy[1:-1]:
        i = 0;  p = m(i,j);  b[p] = 0   # Boundary
        for i in Ix[1:-1]:              # Interior points
            p = m(i,j)
            b[p] = u_1[i,j] + \
              (1-theta)*(
              Fx*(u_1[i+1,j] - 2*u_1[i,j] + u_1[i-1,j]) +\
              Fy*(u_1[i,j+1] - 2*u_1[i,j] + u_1[i,j-1]))\
                + theta*dt*f(i*dx,j*dy,(n+1)*dt) + \
              (1-theta)*dt*f(i*dx,j*dy,n*dt)
        i = Nx;  p = m(i,j);  b[p] = 0  # Boundary
    j = Ny
    for i in Ix:
        p = m(i,j);  b[p] = 0           # Boundary

    # Solve matrix system A*c = b
    c = scipy.linalg.solve(A, b)

    # Fill u with vector c
    for i in Ix:
        for j in Iy:
            u[i,j] = c[m(i,j)]

    # Update u_1 before next step
    u_1, u = u, u_1
```

We use `solve` from `scipy.linalg` and not from `numpy.linalg`. The difference is stated below.

**`scipy.linalg` versus `numpy.linalg`.**

Quote from the SciPy documentation:

`scipy.linalg` contains all the functions in `numpy.linalg` plus some other more advanced ones not contained in `numpy.linalg`.

Another advantage of using `scipy.linalg` over numpy.linalg is that it is always compiled with BLAS/LAPACK support, while for NumPy this is optional. Therefore, the SciPy version might be faster depending on how NumPy was installed.

Therefore, unless you don't want to add SciPy as a dependency to your NumPy program, use `scipy.linalg` instead of `numpy.linalg`.

The code shown above is available in the `solver_dense` function in the file `diffu2D_u0.py`, differing only in the boundary conditions, which in the code can be an arbitrary function along each side of the domain.

We do not bother to look at vectorized versions of filling `A` since a dense matrix is just used of pedagogical reasons for the very first implementation. Vectorization will be treated when `A` has a sparse matrix representation, as in Section 1.6.

**How to debug the computation of $A$ and $b$.**

A good starting point for debugging the filling of $A$ and $b$ is to choose a very coarse mesh, say $N_x = N_y = 2$, where there is just one internal mesh point, compute the equations by hand, and print out `A` and `b` for comparison in the code. If wrong elements in `A` or `b` occur, print out each assignment to elements in `A` and `b` inside the loops and compare with what you expect.

To let the user store, analyze, or visualize the solution at each time level, we include a callback function, named `user_action`, to be called before the time loop and in each pass in that loop. The function has the signature

```
user_action(u, x, xv, y, yv, t, n)
```

where `u` is a two-dimensional array holding the solution at time level `n` and time `t[n]`. The $x$ and $y$ coordinates of the mesh points are given by the arrays `x` and `y`, respectively. The arrays `xv` and `yv` are vectorized representations of the mesh points such that vectorized function evaluations can be invoked. The `xv` and `yv` arrays are defined by

```
xv = x[:,np.newaxis]
yv = y[np.newaxis,:]
```

One can then evaluate, e.g., $f(x, y, t)$ at all internal mesh points at time level `n` by first evaluating $f$ at all points,

```
f_a = f(xv, yv, t[n])
```

and then use slices to extract a view of the values at the internal mesh points: `f_a[1:-1,1:-1]`. The next section features an example on writing a `user_action` callback function.

## 1.5 Verification

A good test example to start with is one that preserves the solution $u = 0$, i.e., $f = 0$ and $I(x, y) = 0$. This trivial solution can uncover some bugs.

The first real test example is based on having an exact solution of the discrete equations. This solution is linear in time and quadratic in space:

$$u(x, y, t) = 5tx(L_x - x)y(y - L_y)\,.$$

Inserting this manufactured solution in the PDE shows that the source term $f$ must be

$$f(x, y, t) = 5x(L_x - x)y(y - L_y) + 10\alpha t(x(L_x - x) + y(y - L_y))\,.$$

We can use the `user_action` function to compare the numerical solution with the exact solution at each time level. A suitable helper function for checking the solution goes like this:

```
def quadratic(theta, Nx, Ny):

    def u_exact(x, y, t):
        return 5*t*x*(Lx-x)*y*(Ly-y)
    def I(x, y):
        return u_exact(x, y, 0)
    def f(x, y, t):
        return 5*x*(Lx-x)*y*(Ly-y) + 10*a*t*(y*(Ly-y)+x*(Lx-x))

    # Use rectangle to detect errors in switching i and j in scheme
    Lx = 0.75
    Ly = 1.5
    a = 3.5
    dt = 0.5
    T = 2

    def assert_no_error(u, x, xv, y, yv, t, n):
        """Assert zero error at all mesh points."""
        u_e = u_exact(xv, yv, t[n])
        diff = abs(u - u_e).max()
        tol = 1E-12
        msg = 'diff=%g, step %d, time=%g' % (diff, n, t[n])
        print msg
        assert diff < tol, msg

    solver_dense(
        I, a, f, Lx, Ly, Nx, Ny,
        dt, T, theta, user_action=assert_no_error)
```

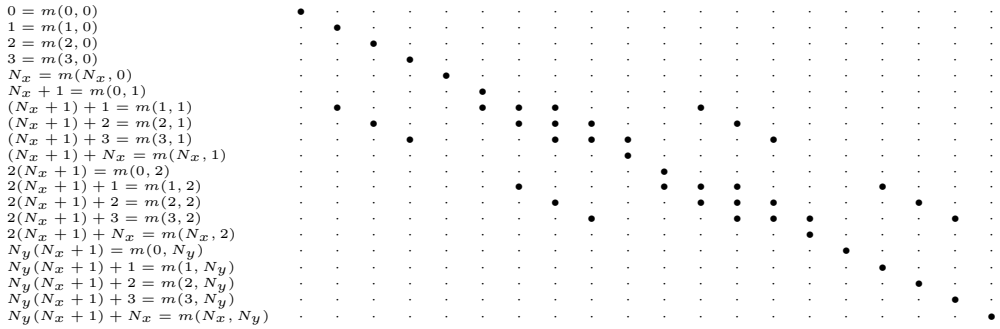
A true test function for checking the quadratic solution for several different meshes and $\theta$ values can take the form

```
def test_quadratic():
    # For each of the three schemes (theta = 1, 0.5, 0), a series of
    # meshes are tested (Nx > Ny and Nx < Ny)
    for theta in [1, 0.5, 0]:
        for Nx in range(2, 6, 2):
            for Ny in range(2, 6, 2):
                print 'testing for %dx%d mesh' % (Nx, Ny)
                quadratic(theta, Nx, Ny)
```

## 1.6 Implementation with a sparse coefficient matrix

We used a sparse matrix implementation in Section **??** for a 1D problem with a tridiagonal matrix. The present matrix, arising from a 2D problem, has five diagonals, but we can use the same sparse matrix data structure `scipy.sparse.diags`.

**Understanding the diagonals.** Let us look closer at the diagonals in the example with a $4 \times 3$ mesh as depicted in Figure 2 and its associated matrix visualized by dots for zeros and bullets for nonzeros. From the example mesh, we may generalize to an $N_x \times N_y$ mesh.




The main diagonal has $N = (N_x+1)(N_y+1)$ elements, while the sub- and super-diagonals have $N-1$ elements. By looking at the matrix above, we realize that the lower diagonal starts in row $N_x+1$ and goes to row $N$, so its length is $N-(N_x+1)$. Similarly, the upper diagonal starts at row 0 and lasts to row $N-(N_x+1)$, so it has the same length. Based on this information, we declare the diagonals by

```
main   = np.zeros(N)            # diagonal
lower  = np.zeros(N-1)          # subdiagonal
upper  = np.zeros(N-1)          # superdiagonal
lower2 = np.zeros(N-(Nx+1))     # lower diagonal
upper2 = np.zeros(N-(Nx+1))     # upper diagonal
b      = np.zeros(N)            # right-hand side
```

**Filling the diagonals.** We run through all mesh points and fill in elements on the various diagonals. The line of mesh points corresponding to $j = 0$ are all on the boundary, and only the main diagonal gets a contribution:

```
m = lambda i, j: j*(Nx+1) + i
j = 0; main[m(0,j):m(Nx+1,j)] = 1  # j=0 boundary line
```

Then we run through all interior $j = \text{const}$ lines of mesh points. The first and the last point on each line, $i = 0$ and $i = N_x$, correspond to boundary points:

```
for j in Iy[1:-1]:              # Interior mesh lines j=1,...,Ny-1
    i = 0;   main[m(i,j)] = 1
    i = Nx;  main[m(i,j)] = 1  # Boundary
```

For the interior mesh points $i = 1, \ldots, N_x - 1$ on a mesh line $y = \text{const}$ we can start with the main diagonal. The entries to be filled go from $i = 1$ to $i = N_x - 1$ so the relevant slice in the `main` vector is `m(1,j):m(Nx,j)`:

```
main[m(1,j):m(Nx,j)] = 1 + 2*theta*(Fx+Fy)
```

The `upper` array for the superdiagonal has its index 0 corresponding to row 0 in the matrix, and the array entries to be set go from $m(1, j)$ to $m(N_x - 1, j)$:

```
upper[m(1,j):m(Nx,j)] = - theta*Fx
```

The subdiagonal (`lower` array), however, has its index 0 corresponding to row 1, so there is an offset of 1 in indices compared to the matrix. The first nonzero occurs (interior point) at a mesh line $j = \text{const}$ corresponds to matrix row $m(1, j)$, and the corresponding array index in `lower` is then $m(1, j)$. To fill the entries from $m(1, j)$ to $m(N_x - 1, j)$ we set the following slice in `lower`:

```
lower_offset = 1
lower[m(1,j)-lower_offset:m(Nx,j)-lower_offset] = - theta*Fx
```

For the upper diagonal, its index 0 corresponds to matrix row 0, so there is no offset and we can set the entries correspondingly to `upper`:

```
upper2[m(1,j):m(Nx,j)] = - theta*Fy
```

The `lower2` diagonal, however, has its first index 0 corresponding to row $N_x + 1$, so here we need to subtract the offset $N_x + 1$:

```
lower2_offset = Nx+1
lower2[m(1,j)-lower2_offset:m(Nx,j)-lower2_offset] = - theta*Fy
```

We can now summarize the above code lines for setting the entries in the sparse matrix representation of the coefficient matrix:

```
lower_offset = 1
lower2_offset = Nx+1
m = lambda i, j: j*(Nx+1) + i

j = 0; main[m(0,j):m(Nx+1,j)] = 1  # j=0 boundary line
for j in Iy[1:-1]:              # Interior mesh lines j=1,...,Ny-1
    i = 0;   main[m(i,j)] = 1  # Boundary
    i = Nx;  main[m(i,j)] = 1  # Boundary
    # Interior i points: i=1,...,N_x-1
    lower2[m(1,j)-lower2_offset:m(Nx,j)-lower2_offset] = - theta*Fy
    lower[m(1,j)-lower_offset:m(Nx,j)-lower_offset] = - theta*Fx
    main[m(1,j):m(Nx,j)] = 1 + 2*theta*(Fx+Fy)
    upper[m(1,j):m(Nx,j)] = - theta*Fx
    upper2[m(1,j):m(Nx,j)] = - theta*Fy
j = Ny; main[m(0,j):m(Nx+1,j)] = 1  # Boundary line
```

The next task is to create the sparse matrix from these diagonals:

```
import scipy.sparse

A = scipy.sparse.diags(
    diagonals=[main, lower, upper, lower2, upper2],
    offsets=[0, -lower_offset, lower_offset,
             -lower2_offset, lower2_offset],
    shape=(N, N), format='csr')
```

**Filling the right-hand side; scalar version.** Setting the entries in the right-hand side is easier since there are no offsets in the array to take into account. The is in fact similar to the one previously shown when we used a dense matrix representation (the right-hand side vector is, of course, independent of what type of representation we use for the coefficient matrix). The complete time loop goes as follows.

```
import scipy.sparse.linalg

for n in It[0:-1]:
    # Compute b
    j = 0
    for i in Ix:
        p = m(i,j);  b[p] = 0                       # Boundary
    for j in Iy[1:-1]:
        i = 0;  p = m(i,j);  b[p] = 0               # Boundary
        for i in Ix[1:-1]:
            p = m(i,j)                              # Interior
            b[p] = u_1[i,j] + \
              (1-theta)*(
              Fx*(u_1[i+1,j] - 2*u_1[i,j] + u_1[i-1,j]) +\
              Fy*(u_1[i,j+1] - 2*u_1[i,j] + u_1[i,j-1]))\
                + theta*dt*f(i*dx,j*dy,(n+1)*dt) + \
              (1-theta)*dt*f(i*dx,j*dy,n*dt)
        i = Nx;  p = m(i,j);  b[p] = 0              # Boundary
    j = Ny
    for i in Ix:
        p = m(i,j);  b[p] = 0                       # Boundary

    # Solve matrix system A*c = b
    c = scipy.sparse.linalg.spsolve(A, b)

    # Fill u with vector c
    for i in Ix:
        for j in Iy:
            u[i,j] = c[m(i,j)]

    # Update u_1 before next step
    u_1, u = u, u_1
```

**Filling the right-hand side; vectorized version.** Since we use a sparse matrix and try to speed up the computations, we should examine the loops and see if some can be easily removed by vectorization. In the filling of $A$ we have already used vectorized expressions at each $j = \text{const}$ line of mesh points. We can very easily do the same in the code above and remove the need for loops over the `i` index:

```
for n in It[0:-1]:
    # Compute b, vectorized version

    # Precompute f in array so we can make slices
    f_a_np1 = f(xv, yv, t[n+1])
    f_a_n   = f(xv, yv, t[n])

    j = 0; b[m(0,j):m(Nx+1,j)] = 0     # Boundary
    for j in Iy[1:-1]:
        i = 0;   p = m(i,j);  b[p] = 0 # Boundary
        i = Nx;  p = m(i,j);  b[p] = 0 # Boundary
        imin = Ix[1]
```

```
        imax = Ix[-1]  # for slice, max i index is Ix[-1]-1
        b[m(imin,j):m(imax,j)] = u_1[imin:imax,j] + \
              (1-theta)*(Fx*(
          u_1[imin+1:imax+1,j] -
        2*u_1[imin:imax,j] + \
          u_1[imin-1:imax-1,j]) +
                           Fy*(
          u_1[imin:imax,j+1] -
        2*u_1[imin:imax,j] +
          u_1[imin:imax,j-1])) + \
            theta*dt*f_a_np1[imin:imax,j] + \
          (1-theta)*dt*f_a_n[imin:imax,j]
    j = Ny;  b[m(0,j):m(Nx+1,j)] = 0 # Boundary

    # Solve matrix system A*c = b
    c = scipy.sparse.linalg.spsolve(A, b)

    # Fill u with vector c
    u[:,:] = c.reshape(Ny+1,Nx+1).T

    # Update u_1 before next step
    u_1, u = u, u_1
```

The most tricky part of this code snippet is the loading of values in the one-dimensional array `c` into the two-dimensional array `u`. With our numbering of unknowns from left to right along "horizontal" mesh lines, the correct re-ordering of the one-dimensional array `c` as a two-dimensional array requires first a reshaping as an `(Ny+1,Nx+1)` two-dimensional array and then taking the transpose. The result is an `(Nx+1,Ny+1)` array compatible with `u` both in size and appearance of the function values.

The `spsolve` function in `scipy.sparse.linalg` is an efficient version of Gaussian elimination suited for matrices described by diagonals. Actually, only the matrix elements within the bands (from `lower2` to `upper2`) are computed with, and these elements constitute only a fraction of all $N^2$ matrix elements, a crucial property for large $N$. The Gaussian elimination algorithm for banded matrices is therefore much faster and requires much less storage than standard Gaussian elimination for a dense matrix. More precisely, with $b = N_x + 1$ as the *bandwidth* of the matrix [[[

**hpl 2**: Problem: if $N_x \gg N_y$ one should number the unknowns in $y$ direction to get a smaller bandwidth.

The complete code is found in the `solver_sparse` function in the file `diffu2D_u0.py`.

## 1.7 The Jacobi iterative method

So far we have created a linear system $Ac = b$ by creating the matrix and right-hand side and solved the system by calling an exact algorithm based on Gaussian elimination. A much simpler implementation, which requires no memory for the coefficient matrix $A$, arises if we solve the system by *iterative* methods. These are only approximate, and the core algorithm is repeated many times until the solution converges.

To illustrate the idea, we simply the numerical scheme to the Backward Euler case, $\theta = 1$, so there are fewer terms to write:

$$u_{i,j}^{n+1} - \left(F_x(u_{i-1,j}^{n+1} - 2_{i,j}^{n+1} + u_{i,j}^{n+1}) + F_y(u_{i,j-1}^{n+1} - 2_{i,j}^{n+1} + u_{i,j+1}^{n+1})\right) = + \\ u_{i,j}^{n} + \Delta t f_{i,j}^{n+1} \tag{11}$$

The idea of the *Jacobi* iterative method is to introduce an iteration, here with index $r$, where we in each iteration treat $u_{i,j}^{n+1}$ as unknown, but use values from the previous iteration for the other unknowns $u_{i\pm1,j\pm1}^{n+1}$. Let $u_{i,j}^{n+1,r}$ be the approximation to $u_{i,j}^{n+1}$ in iteration $r$, for all relevant $i$ and $j$ indices. We first solve with respect to $u_{i,j}^{n+1}$ to get the equation to solve:

$$u_{i,j}^{n+1} = (1 + 2F_x + 2F_y)^{-1}\left(F_x(u_{i-1,j}^{n+1} + u_{i,j}^{n+1}) + F_y(u_{i,j-1}^{n+1} + u_{i,j+1}^{n+1})\right) + \\ u_{i,j}^{n} + \Delta t f_{i,j}^{n+1} \tag{12}$$

The iteration is introduced by using iteration index $r$, for computed values, on the right-hand side and $r$ (unknown in this iteration) on the left-hand side:

$$u_{i,j}^{n+1,r+1} = (1 + 2F_x + 2F_y)^{-1}((F_x(u_{i-1,j}^{n+1,r} + u_{i,j}^{n+1,r}) + F_y(u_{i,j-1}^{n+1,r} + u_{i,j+1}^{n+1,r})) + \\ u_{i,j}^{n} + \Delta t f_{i,j}^{n+1}) \tag{13}$$

We start the iteration with the value at the previous time level:

$$u_{i,j}^{n+1,0} = u_{i,j}^{n}, \quad i = 0, \ldots, N_x,\ j = 0, \ldots, N_y\,. \tag{14}$$

A common technique in iterative methods is to introduce a *relaxation*, which means that the new approximation is a weighted mean of the approximation as suggested by the algorithm and the previous approximation. Naming the quantity on the left-hand side of (13) as $u_{i,j}^{n+1,*}$, a new approximation based on relaxation reads

$$u^{n+1,r+1} = \omega u_{i,j}^{n+1,*} + (1-\omega)u_{i,j}^{n+1,r}\,. \tag{15}$$

Under-relaxation means $\omega < 1$, while over-relaxation has $\omega > 1$.

The iteration can be stopped when the change from one iteration to the next is sufficiently small ($\epsilon$), using either an infinity norm,

$$\max_{i,j}\left|u_{i,j}^{n+1,r+1} - u_{i,j}^{n+1,r}\right| \leq \epsilon, \tag{16}$$

or an $L^2$ norm,

$$\left(\Delta x \Delta y \sum_{i,j}(u_{i,j}^{n+1,r+1} - u_{i,j}^{n+1,r})^2\right)^{\frac{1}{2}} \leq \epsilon\,. \tag{17}$$

**hpl 3**: Residual criterion?

To make the mathematics as close as possible to what we will write in a computer, we may introduce some new notation: $u_{i,j}$ is a short notation for $u^{n+1,r+1}_{i,j}$, $u^-_{i,j}$ is a short notation for $u^{n+1,r}_{i,j}$, and $u^{(s)}_{i,j}$ denotes $u^{n+1-s}_{i,j}$. That is, $u_{i,j}$ is the unknown, $u^-_{i,j}$ is its most recently computed approximation, and $s$ counts time levels backwards in time. The Jacobi method (ref(13)) takes the following form with the new notation:

$$u_{i,j} = (1 + 2F_x + 2F_y)^{-1}((F_x(u^-_{i-1,j} + u^-_{i,j}) + F_y(u^{n+1,r}_{i,j-1} + u^{n+1,r}_{i,j+1}))+ u^{(1)}_{i,j} + \Delta t f^{n+1}_{i,j}) \tag{18}$$

We can also quite easily introduce the $\theta$ rule for discretization in time and write up the Jacobi iteration in that case as well:

$$u_{i,j} = (1 + \theta(2F_x + 2F_y))^{-1}(\theta(F_x(u^-_{i-1,j} + u^-_{i,j}) + F_y(u^{n+1,r}_{i,j-1} + u^{n+1,r}_{i,j+1}))+ u^{(1)}_{i,j} + \theta\Delta t f^{n+1}_{i,j}) + (1 - theta)\Delta t f^n_{i,j}+ + (1-\theta)(Fx(u^{(1)}_{i-1,j} - 2u^{(1)}_{i,j} + u^{(1)}_{i+1,j}) + Fy(u^{(1)}_{i,j-1} - 2u^{(1)}_{i,j} + u^{(1)}_{i,j+1}))\,. \tag{19}$$

## 1.8 Implementation of the Jacobi iterative method

The Jacobi method needs to coefficient matrix or right-hand side vector, but it needs an array for $u$ in the previous iteration. We call this array `u_` (using the notation at the end of the previous section), the unknown ais `u`, and `u_1` is, as usual, the computed solution one time level back in time. With a $\theta$ rule in time, the time loop can be coded like this:

```
for n in It[0:-1]:
    # Solve linear system by Jacobi iteration at time level n+1
    u_[:,:] = u_1  # Start value
    converged = False
    r = 0
    while not converged:
        if version == 'scalar':
            j = 0
            for i in Ix:
                u[i,j] = U_0y(t[n+1])           # Boundary
            for j in Iy[1:-1]:
                i = 0;   u[i,j] = U_0x(t[n+1])  # Boundary
                i = Nx;  u[i,j] = U_Lx(t[n+1])  # Boundary
                # Interior points
                for i in Ix[1:-1]:
                    u_new = 1.0/(1.0 + 2*Fx + 2*Fy)*(theta*(
                        Fx*(u_[i+1,j] + u_[i-1,j]) +
                        Fy*(u_[i,j+1] + u_[i,j-1])) + \
                    u_1[i,j] + \
                    (1-theta)*(Fx*(
                    u_1[i+1,j] - 2*u_1[i,j] + u_1[i-1,j]) +
                      Fy*(
```

```
                u_1[i,j+1] - 2*u_1[i,j] + u_1[i,j-1]))\
                  + theta*dt*f(i*dx,j*dy,(n+1)*dt) + \
                (1-theta)*dt*f(i*dx,j*dy,n*dt))
                u[i,j] = omega*u_new + (1-omega)*u_[i,j]
        j = Ny
        for i in Ix:
            u[i,j] = U_Ly(t[n+1])      # Boundary
    elif version == 'vectorized':
        j = 0;  u[:,j] = U_0y(t[n+1])  # Boundary
        i = 0;  u[i,:] = U_0x(t[n+1])  # Boundary
        i = Nx; u[i,:] = U_Lx(t[n+1])  # Boundary
        j = Ny; u[:,j] = U_Ly(t[n+1])  # Boundary
        # Internal points
        f_a_np1 = f(xv, yv, t[n+1])
        f_a_n   = f(xv, yv, t[n])
        u_new = 1.0/(1.0 + 2*Fx + 2*Fy)*(theta*(Fx*(
          u_[2:,1:-1] + u_[:-2,1:-1]) +
            Fy*(
          u_[1:-1,2:] + u_[1:-1,:-2])) +\
        u_1[1:-1,1:-1] + \
          (1-theta)*(Fx*(
          u_1[2:,1:-1] - 2*u_1[1:-1,1:-1] + u_1[:-2,1:-1]) +\
            Fy*(
          u_1[1:-1,2:] - 2*u_1[1:-1,1:-1] + u_1[1:-1,:-2]))\
          + theta*dt*f_a_np1[1:-1,1:-1] + \
          (1-theta)*dt*f_a_n[1:-1,1:-1])
        u[1:-1,1:-1] = omega*u_new + (1-omega)*u_[1:-1,1:-1]
    r += 1
    converged = np.abs(u-u_).max() < tol or r >= max_iter
    u_[:,:] = u

# Update u_1 before next step
u_1, u = u, u_1
```