SEMT30002 Scientific Computing and Optimisation

Week 5 Demos: The 2D Poisson equation

Matthew Hennessy

The demos for this week will focus on

- Solving the 2D Poisson equation
- Using memory profiling to explore the benefits of sparse matrices

We start by importing some packages

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Example 1 - solving the 2D Poisson equation

In this example, we will solve the PDE

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + 1 = 0$$

on the rectangular domain $a \leq x \leq b$ and $c \leq y \leq d$. We will assume that $u = 0$ on all of the boundaries.

The for solving this problem is on the unit website - we'll walk through this code now

Preliminaries

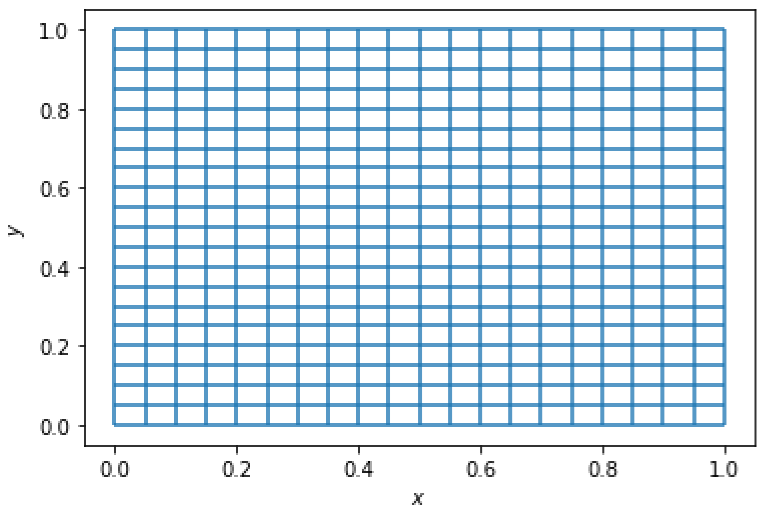
We define parameters associated with the spatial discretisation:

In [3]:
```
# domain parameters
a = 0; b = 1
c = 0; d = 1

# number of grid points
Nx = 20; Ny = 20

# grid points
```

```
x = np.linspace(0, 1, Nx+1)
y = np.linspace(0, 1, Ny+1)
x_int = x[1:Nx]
y_int = y[1:Ny]

# grid spacing in x and y directions
dx = (b - a) / Nx
dy = (d - c) / Ny
```

We now visualise the 2D grid:

In [4]:
```
plt.hlines(y, a, b)
plt.vlines(x, c, d)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()
```



- It's useful to calculate the total number of unknowns, which are called degrees of freedom (dof)
- This will give us an indication of how large the linear system will be

In [5]:
```
# total number of unknowns
dof = (Nx - 1) * (Ny - 1)
print('There are', dof, 'unknowns to solve for')
```

```
There are 361 unknowns to solve for
```

We now set the constant value of the source term

In [6]:
```
# Value of the constant source term
q = 1
```

Formulating the linear system

- Now that the preliminary variables have been defined, we now look at constructing the linear system of algebraic equations that will be solved
- We'll solve this system using SciPy's `root` function
- We need to formulate the algebraic system as a vector equation of the form $\boldsymbol{F}(\boldsymbol{U}) = \boldsymbol{0}$, where $\boldsymbol{U}$ is the 1D solution vector

The approach

1. Define a Python function that takes as input a 1D solution vector $\boldsymbol{U}$ with components U_k
2. Convert the 1D solution vector U_k into a 2D solution array with components $u_{i,j}$
3. Use a double `for` loop to evaluate the discrete Poisson equation

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

4. Handle the edge/corner cases with `if` and `elif` statements
5. Convert the 2D solution array $f_{i,j}$ into a 1D array F_k
6. Return the 1D array $\boldsymbol{F}$ with components F_k

Mapping local indices to global indices

- We need to define a function that converts local indices, i and j, into a global index k
- This is needed to transform U_k to $u_{i,j}$ and vice-versa

In [7]:
```
# mapping from grid indices (i,j) to global indices (k)
k = lambda i,j : i + (Nx - 1) * j
```

Using local and global indices

- To use the `root` function, we need to provide an initial guess of the solution
- We will use $u_0 = xy(1-x)(1-y)$ as the initial guess
- We first create a 2D array and then convert this into a 1D array

In [8]:
```
# pre-allocate the 2D array
u_0 = np.zeros((Nx - 1, Ny - 1))

# use a double for loop to create the initial guess as a 2D array
for i in range(Nx - 1):
    for j in range(Ny - 1):
        u_0[i, j] = x_int[i] * y_int[j] * (1 - x_int[i]) * (1 - y_int[j])
```

Now that the 2D array has been created, we convert it to a 1D array as follows:

In [9]:
```python
# pre-allocate the 1D array
U_0 = np.zeros((Nx - 1) * (Ny - 1))

# use a double for loop to store each u[i,j] in U_k
for i in range(Nx - 1):
    for j in range(Ny - 1):
        U_0[k(i,j)] = u_0[i,j]
```

Constructing the 2D array `f[i,j]`

- We'll now look at how to build the 2D array that stores the discrete Poisson equation

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

- We'll go through the code that is on the unit website for this

Solving the system

- The algebraic system $\boldsymbol{F}(\boldsymbol{U}) = \boldsymbol{0}$ can be solved using `root`

```python
sol = root(dirichlet_problem, U_0)
```

- Don't forget to use the `sol.success` attribute to check for convergence
- The 1D solution array (in `sol.x`) can then be converted into a 2D array and then visualised

Example 2 - Sparse vs dense

- With 2D problems, the number of unknowns to be solved for increases very rapidly as the grid is refined.
    - This is known as **the curse of dimensionality**
- This will have a major impact on the memory required to find a solution and the time needed to compute a solution
- Sparse matrices can lead to significant increases in performance and decreases in memory use

- To illustrate the benefit of sparse matrices, we'll now solve a linear system given by $\mathbf{A}\boldsymbol{u} = \boldsymbol{b}$ where the matrix $\mathbf{A}$ is just the identity matrix
- Both dense and sparse matrices will be used
- We'll examine the memory cost using memory profiling

- The code for this demo can be downloaded from Week 5 of the unit website
- When running this code, you should see that using sparse matrices leads to a 1000-fold reduction in memory!
  - The most memory-intensive step is solving the linear system
  - 740 MB used with dense matrices, only 0.4 MB used with sparse matrices
- The code also runs much faster when sparse matrices are used

Example 3 - Memory profiling with Jupyter notebooks

- Memory profiling within Jupyter notebooks is simple, but the info you get is very basic
- First load the memory profiling extension

In [10]:
```
%load_ext memory_profiler
```

- Now define the Python function you want to profile as usual, e.g.

In [11]:
```
def solve_linear_system():
    N = 1000
    b = np.random.random(N)
    A = np.random.random((N, N))
    u = np.linalg.solve(A, b)
```

- Now use the `memit` magic function

In [12]:
```
%memit solve_linear_system()
```

```
peak memory: 102.04 MiB, increment: 17.17 MiB
```

- From the increment value, we see that 18 MB of memory was needed to solve this problem

- As you can see, the memory information is very basic
- There is supposedly a way to get line-by-line memory info, but I've never gotten this to work