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

The code for solving the 2D Poisson equation in the demos will be contained on Python modules I've created. The exercises this week will focus on developing your own code for the 2D Poisson equation.

We start by importing these modules and associated packages

In [1]:
```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

# Below are modules I've created
import finite_diff as fd
from pde_solvers import Poisson2D
```

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.

We start by defining parameters associated with the spatial discretisation:

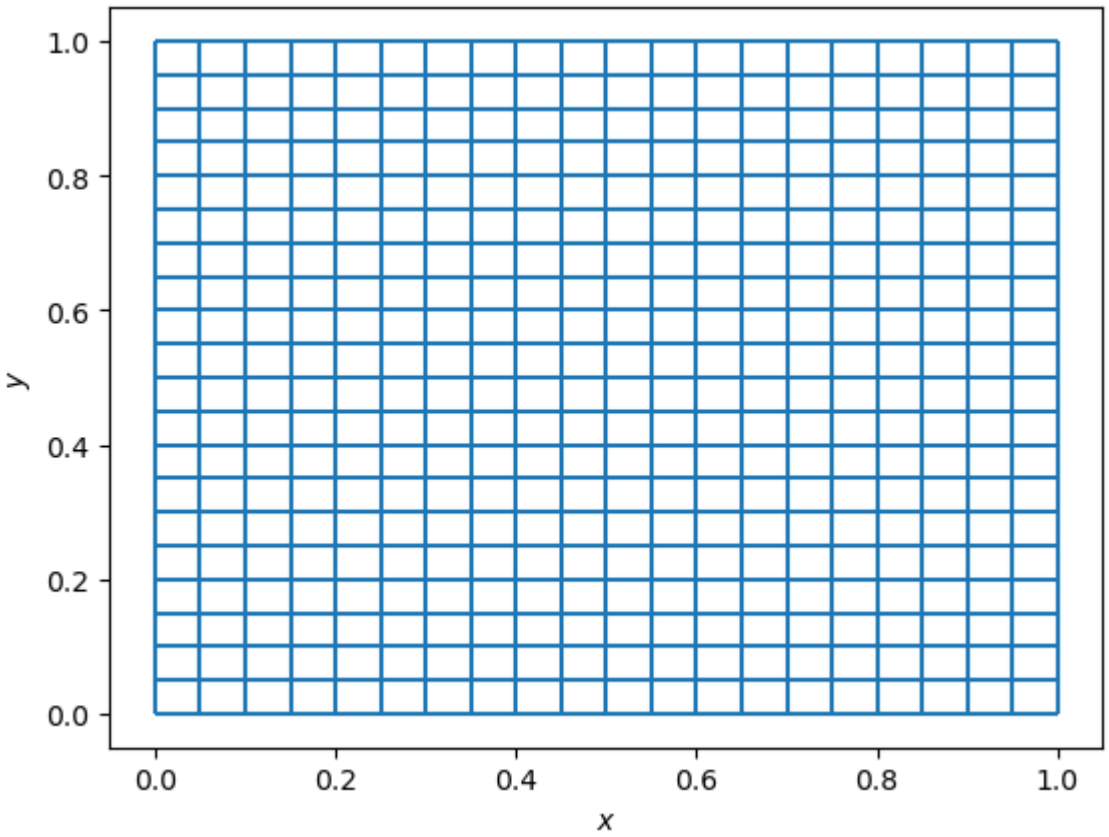
In [2]:
```
# Spatial domain
a = 0
b = 1
```

```
c = 0
d = 1

# Spatial discretisation
Nx = 20
Ny = 20

"""
Create the grid (with all grid points).  This just creates two 1D NumPy arra
"""
grid = fd.SpaceGrid2D(Nx, a, b, Ny, c, d)
print(grid.x)
print(grid.y)
```

```
[0.   0.05 0.1  0.15 0.2  0.25 0.3  0.35 0.4  0.45 0.5  0.55 0.6  0.65
 0.7  0.75 0.8  0.85 0.9  0.95 1.  ]
[0.   0.05 0.1  0.15 0.2  0.25 0.3  0.35 0.4  0.45 0.5  0.55 0.6  0.65
 0.7  0.75 0.8  0.85 0.9  0.95 1.  ]
```

We we'll now visualise the 2D grid:

In [3]:
```
plt.hlines(grid.y, a, b)
plt.vlines(grid.x, c, d)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()
```



The next step is to define the four Dirichlet boundary conditions to be imposed at (i) $x = a$; (ii) $x = b$; (iii) $y = c$; (iv) $y = d$. Python's lambda functions are used to evaluate

the boundary conditions at each grid point on the boundary:

In [4]:
```
bcs = {
    "left": fd.BoundaryCondition('Dirichlet', lambda y: 0),
    "right": fd.BoundaryCondition('Dirichlet', lambda y: 0),
    "top": fd.BoundaryCondition('Dirichlet', lambda x: 0),
    "bottom": fd.BoundaryCondition('Dirichlet', lambda x: 0)
}
```

Since we have Dirichlet boundary conditions, we trim the NumPy arrays with the x and y grid points to remove the exterior grid points:

In [5]:
```
grid.x = grid.x[1:-1]
grid.y = grid.y[1:-1]
```

The next step is to define a Python function that evaluates the source term. We allow the source term to depend on some parameters for generality:

In [6]:
```
def q(x, y, pars):
    return 1
```

Using the `Poisson2D` class, the discretised PDE is generated and stored in an object:

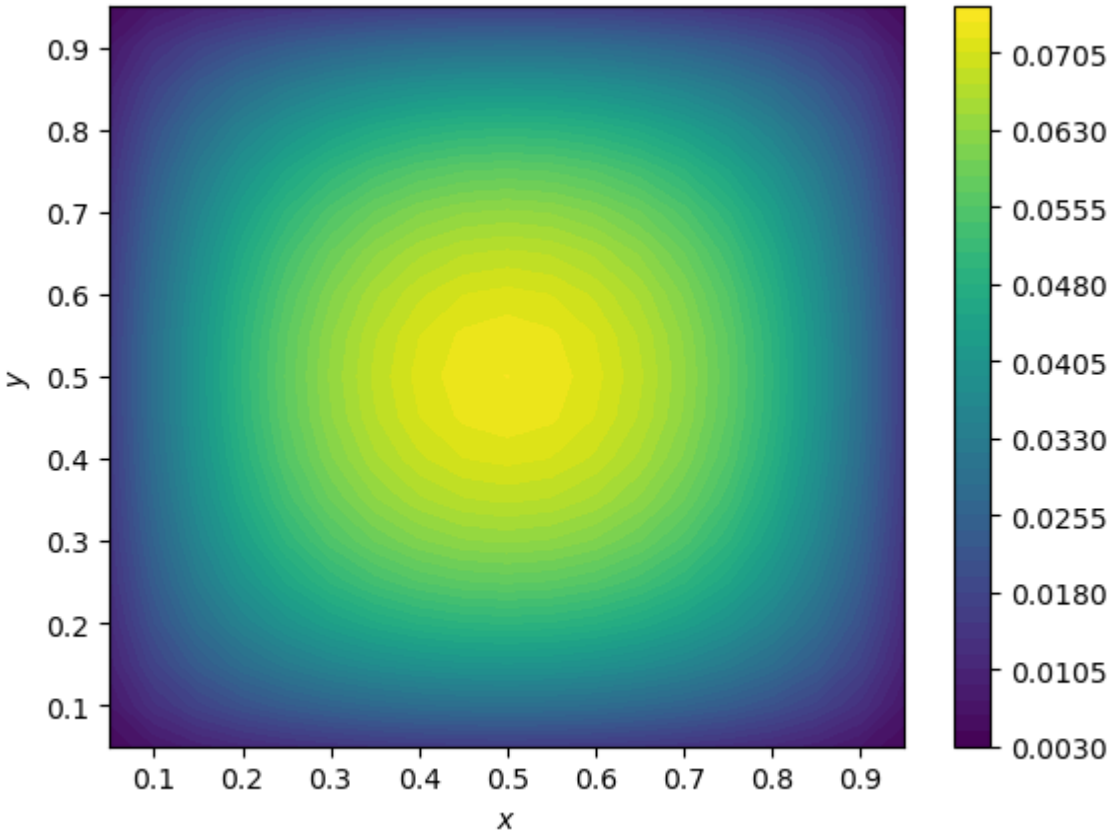
In [7]:
```
pde = Poisson2D(grid, bcs, q = q)
```

We now solve the problem using dense matrices:

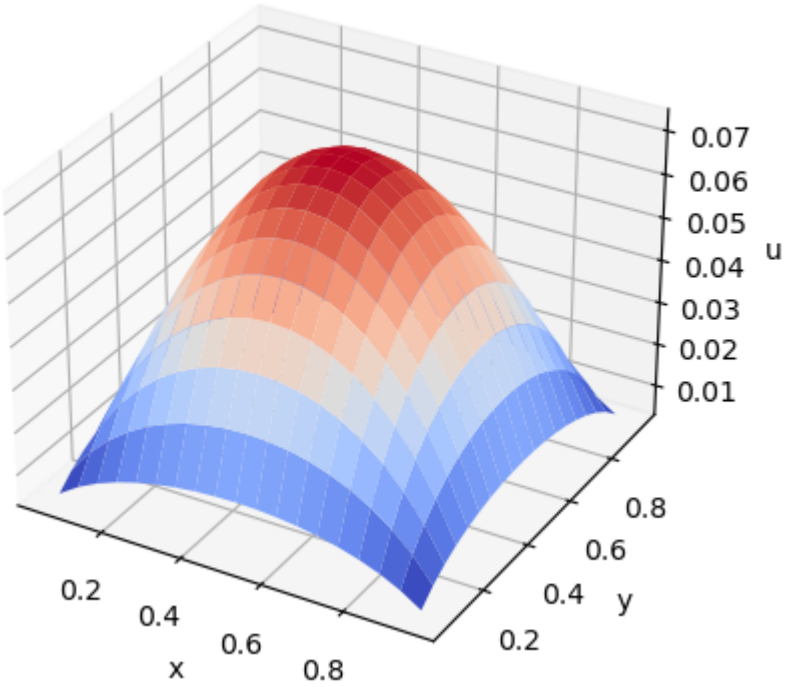
In [8]:
```
u = pde.solve(matrix_type = 'dense')
```

The solution can be visualised in a number of ways. We'll use filled contour plots and surface plots:

In [9]:
```
plt.contourf(grid.x, grid.y, u.T, 50)
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.colorbar()
plt.show()
```



In [10]:
```
# Turn the 1D x,y arrays into 2D arrays
xx, yy = np.meshgrid(grid.x, grid.y)

# Now create a surface plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(xx, yy, u.T, cmap = cm.coolwarm)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("u")
plt.show()
```



Example 2 - Sparse vs dense

It was very fast to solve the previous problem because only a small number of grid points were used. However, with 2D problems, the number of unknowns to be solved for increases very rapidly as the grid is refined. This will have a major impact on the memory required to find a solution if dense matrices are used.

For example, let's solve the previous problem using a 101×101 grid. This corresponds to $N_x = N_y = 100$. Since we have Dirichlet boundary conditions, we only need to compute the solution at the interior grid points. The number of unknowns, also called the degree of freedoms (dofs), is then:

In [11]:
```
Nx = 100
Ny = 100

dof = (Nx - 2) * (Ny - 2)
print(f'There are {dof} unknowns to solve for')
```

```
There are 9604 unknowns to solve for
```

This means that the matrix in the discretised problem (linear system of equations) will be of size 9604×9604. The amount of memory required to store this dense matrix will be

In [12]:
```
print('Memory required to store dense matrix:', dof**2 * 8 / 1e6, 'MB')
```

```
Memory required to store dense matrix: 737.894528 MB
```

Even more memory will be needed to store the discrete source term, boundary condition vector, and solve the linear system.

In Jupyter Notebooks, we can easily profile memory usage. We first load the `memory_profiler` package as follows:

In [13]:
```
%load_ext memory_profiler
```

Now we update the grid:

In [14]:
```
grid = fd.SpaceGrid2D(Nx, a, b, Ny, c, d, bcs)
```

Now we define a function that contains the code we want to profile the memory of:

In [15]:
```
def build_and_solve(matrix_type):
    pde = Poisson2D(grid, bcs, q = q)
    pde.solve(matrix_type = matrix_type)
```

Now we use the `memit` magic function

In [16]:
```
%memit build_and_solve('dense')
```

```
peak memory: 962.75 MiB, increment: 813.38 MiB
```

The increment tells us how much memory was used by the function. It took just over 800 MB to build the discretised problem and solve it.

The peak memory describes the maximum **total** memory used Python. This includes all of the memory that has been used before running the function, e.g. memory from loading `matplotlib`, memory used from running the previous example, etc

We can profile the memory when sparse matrices are used:

In [17]:
```
%memit build_and_solve('sparse')
```

```
peak memory: 189.45 MiB, increment: 13.11 MiB
```

Not only did the code run noticeably faster, but it uses about 60 times less memory (about 13 MB was needed in this case).