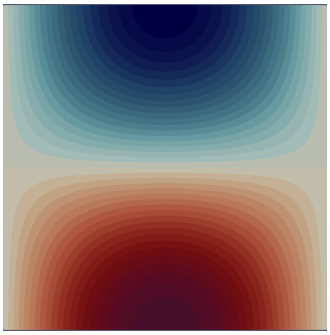
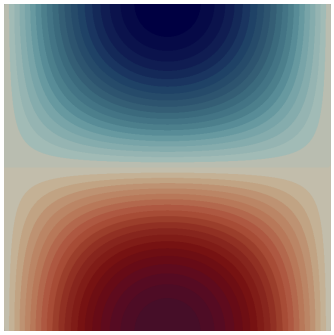
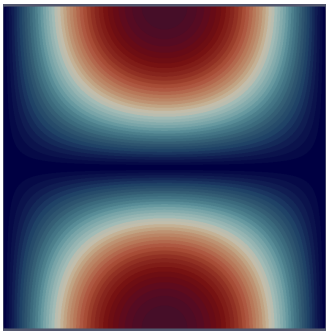
Question 4 (Step 4)
The below shows 2D contour plots for the Firedrake solutions produced for Method 1 (solves from the weak form) and Method 2 (solves from the Ritz-Galerkin principle). Closer analysis of the difference between Method 1 and 2 is in Question 5. The continuous Galerkin family of function spaces is used - this uses continuous linear piecewise polynomial basis functions and is more appropriate for the Poisson equation. Note the order *p* refers to the polynomial degree used for the test and trial functions. A 128x128 mesh ($h = 1/\nabla x$ = 0.0078125) was used initially with order



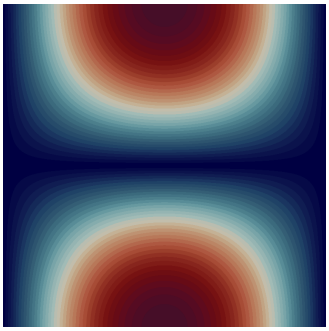
u_h for Method 1
{0.0078125,1}



u_h for Method 2
{0.0078125,1}



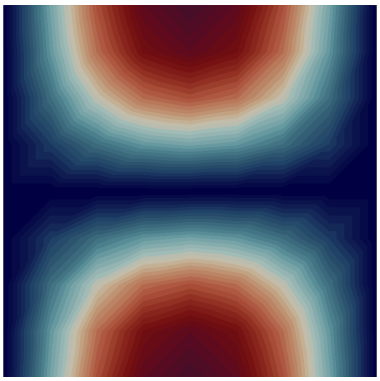
$|u_h - u_e|$ for Method 1
{0.0078125,1}



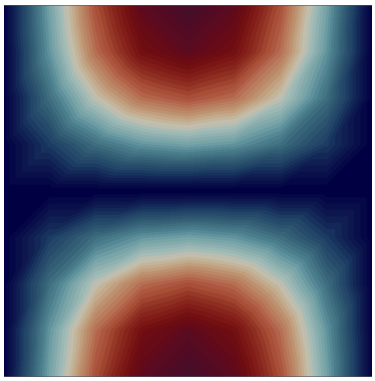
$|u_h - u_e|$ for Method 2
{0.0078125,1}

of accuracy 1. The difference between the numerical solution and the exact solution for each method are shown. The axes are x,y = [0,1] x [0,1] but omitted to save space.

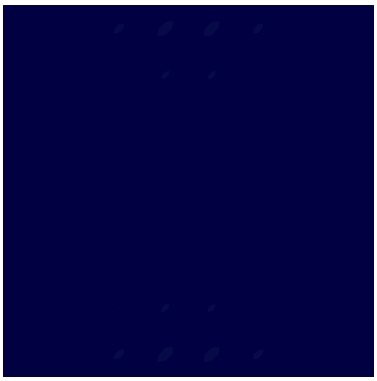
Square meshes with dimensions 4x4 to 1028x1028 (through powers of 2) were tested at *p* = 1, 2, 3, 4. This *{h, p}* selection was chosen as the solutions are computed within a reasonable timeframe (mostly under a minute) but allow for an interesting investigation for optimisation. The following shows $|u_h(x,y) - u_e(x,y)|$ for a select few *h*-refinements and *p*-refinements. Finer



$|u_h - u_e|$ for 8x8 mesh, Method 1 *{0.0125,1}*

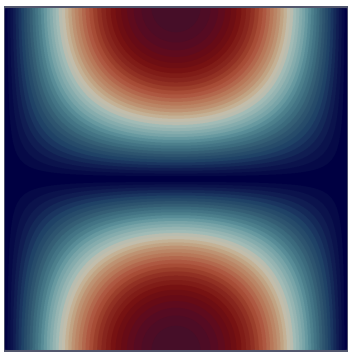


$|u_h - u_e|$ for 8x8 mesh, Method 2 *{0.0125,1}*

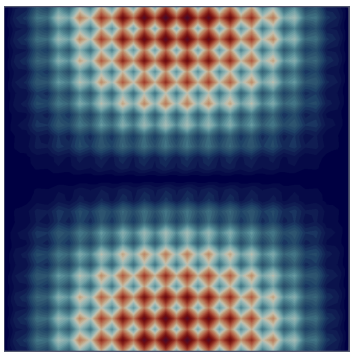


$|u_h - u_e|$ for 8x8 mesh, Method 1 *{0.0125,4}*

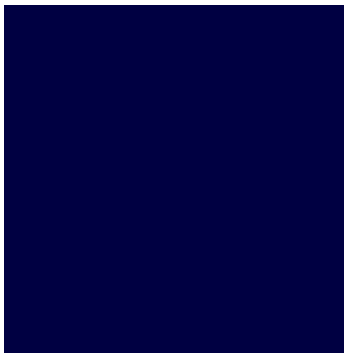
$|u_h - u_e|$ for 8x8 mesh, Method 2 *{0.0125,4}*



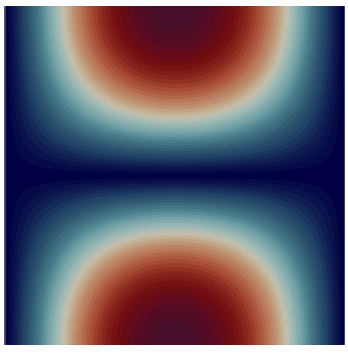
$|u_h - u_e|$ for 526x526 mesh, Method 2
{0.001937984496,2}



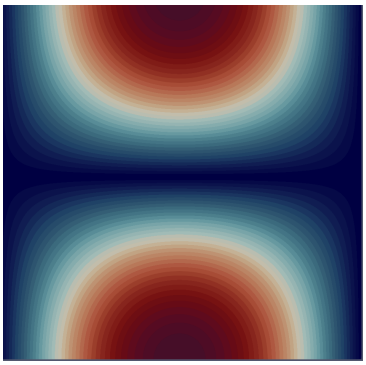
$|u_h - u_e|$ for 16x16 mesh, Method 2
{0.0625,1}

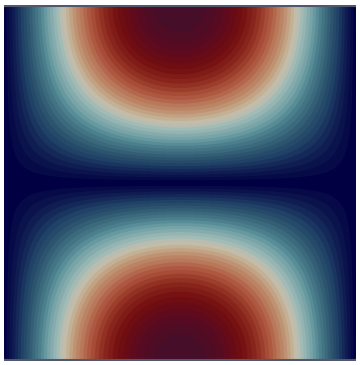


$|u_h - u_e|$ for 32x32 mesh, Method 1
{0.03125,4}



$|u_h - u_e|$ for 2048x2048 mesh, Method 2
{0.00048828125,1}





$|u_h - u_e|$ for 516x516 mesh, Method 1 *{0.001937984496,1}*

$|u_h - u_e|$ for 256x256 mesh, Method 1 *{0.00390625,1}*

meshes and higher orders of accuracy are mostly omitted as $|u_h - u_e|$ seems to be too small to visualise anything interesting - see the 8x8 mesh at *p*=4.

Both Method 1 and 2 are shown as numerical values show the difference between results is minimal (generally to the maximum order of $\sim 10^{-11}$). Visually, the solution contours "smooth out" as accuracy to the exact solution increases. It is more interesting to consider the values of the L^2 norms:

$$\|u_h - u_e\| = \sqrt{\int_\Omega |u_h - u_e|^2 \, d\Omega}$$

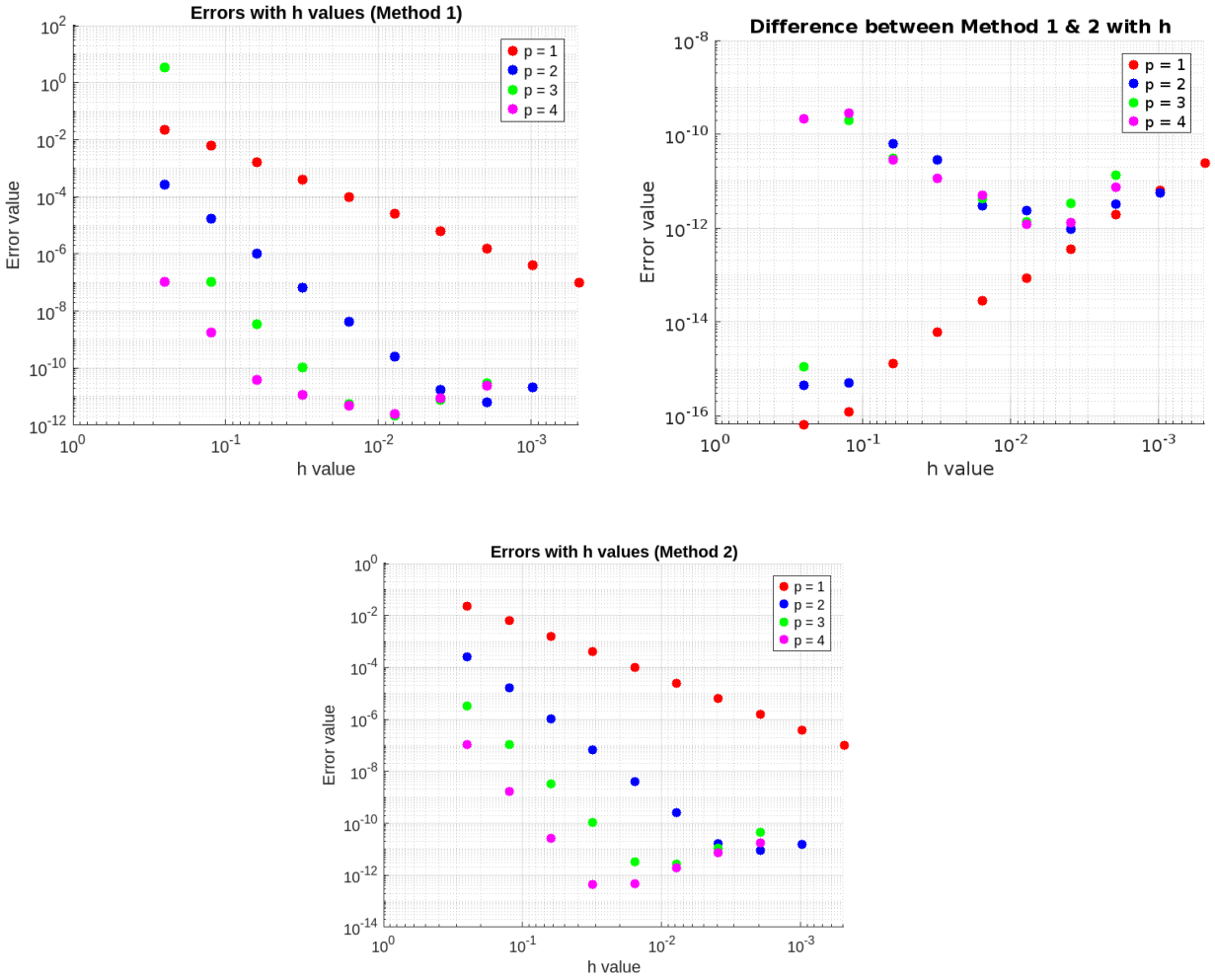
The table below shows the values which are the same to 11 decimal places (colour coded to Method 1).

Mesh Size (NxN)	**h**	**p**	**Meth. 1 L^2 value**	**Meth. 2 L^2 value**	**Meth 1. L^2 (Rounded)**	**Meth 2. L^2 (Rounded)**
128	0.0078125	3	2.19E-12	2.67E-12	0.00E+00	0.00E+00
128	0.0078125	4	2.45E-12	1.89E-12	0.00E+00	0.00E+00
64	0.015625	4	4.94E-12	4.74E-13	0.00E+00	0.00E+00
516	0.001937984	2	6.42E-12	8.97E-12	1.00E-11	1.00E-11
256	0.00390625	3	7.62E-12	1.08E-11	1.00E-11	1.00E-11
64	0.015625	3	5.35E-12	3.36E-12	1.00E-11	0.00E+00
256	0.00390625	4	8.59E-12	7.45E-12	1.00E-11	1.00E-11
32	0.03125	4	1.15E-11	4.32E-13	1.00E-11	0.00E+00
1024	0.000976562	2	2.09E-11	1.54E-11	2.00E-11	2.00E-11
256	0.00390625	2	1.73E-11	1.69E-11	2.00E-11	2.00E-11
516	0.001937984	4	2.49E-11	1.81E-11	2.00E-11	2.00E-11

We determine these meshes are functionally the same, as they produce the same levels of accuracy to a high level. The table above demonstrates that order of accuracy is far more powerful in achieving accurate solutions compared to mesh size. This is an important result when considering computational cost, as fine meshes tend to take far longer to run even at low orders

of accuracy, whereas increasing order of accuracy does not significantly impact run time on coarse meshes. Looking at the table, we can see a 32x32 mesh at *p*=4 (*{0.03125,4}*) produces the same accuracy as a 516x516 mesh at *p*=2 (*{0.001937984496, 2}),* although the latter takes significantly longer to run (>5 minutes compared to <1 minute runtime of the 32x32 mesh). This trend occurs for all segments of the table, that is, there exists a coarse mesh at a higher *p* which produces the same results as finer, more computationally costly meshes at lower *p.*

Furthermore, the largest mesh size tested, 2048x2048 (*{0.00048828125, 1}*) has L^2 ~ 9.81E-08 (for Method 1 and Method 2) at *p*=1, but the minimal L^2 value occurred for Method 2 on a relatively course 32x32 grid at L^2 ~ 4.32E-13 for *p*=4.



The graphs above were created just out of interest to see what visual patterns would emerge as *p,* and *h* change. Note “Error value” on the y-axis refers to the L^2 norm. It is predictable that the error decreases with *h*, and the error values decrease more rapidly as *p* decreases. As $h \to 0$, the error values plateau, which suggests increasing mesh refinement past a 1024x1024 mesh will not yield any higher level of accuracy. We can also visibly see the trend noted in the table above, that is, that higher orders of accuracy tend to achieve their minimum error values for coarser meshes. It would be interesting to increase *p* even further to see if this trend will plateau or we can find a new minimal error. Note $\|u_h - u_e\|$ decreases asymptotically - $\|u_h - u_e\| = 0$ is unlikely even for increased *p.*

Note that the difference between Method 1 and Method 2 tends to decrease as accuracy increases, with the exception of *p*=1 and some outlier values.

Question 5

We first define the mesh and relevant function spaces -

```
V = FunctionSpace(mesh, 'CG', 4)
```

Defines the function space we will take our test and basis functions v and ϕ from, that is, V = $L^2(\Omega)$. In the submitted code, p=4, that is, our basis functions ϕ are polynomials of the 4th degree $\alpha_1 x^4 + \alpha_2 x^3 + \alpha_3 x^2 + \alpha_4 x + \alpha_5$.

Method 1

We start by defining our various functions to be used, that is,

```
u = TrialFunction(V)
v = TestFunction(V)
```

defines the test functions v we will use (corresponding to w_j and δu in written answers) and our approximation of the exact solution (the trial function). And,

```
f = Function(V).interpolate(2*pi**2*sin(pi*x)*cos(pi*y))
```

defines the given function from the Poisson equation $-\nabla^2 u = f = 2\pi^2 \sin(\pi x)\cos(\pi y)$.

Recall the weak form which we derived by hand -

$$\int_\Omega \nabla \delta u \cdot \nabla u \, d\Omega = \int_\Omega \delta u f \, d\Omega$$

We define the left-hand side by `a` and the right-hand side by `L` in the following:

```
a = (inner(grad(u),grad(v)))*dx
L = (f*v)*dx
```

`dx` indicates the integral - it tells Firedrake to integrate over the cells of the mesh, ie, our domain Ω.

Then we define our solution space `u_1`, and the Dirichlet boundary conditions $u(0,y) = u(1,y) = 0$ on $\delta\Omega_D$ -

```
u_1 = Function(V, name='u_1')
bc_x0 = DirichletBC(V, Constant(0), 1)
bc_x1 = DirichletBC(V, Constant(0), 2)
```

Where `V` is the function space, `Constant(0)` sets the value of u on $\delta\Omega_D$, and `1`, `2` are the IDs corresponding to the x = 0, x=1 boundaries of $\delta\Omega$ respectively. We do not need to explicitly state the Neumann boundary conditions $\delta_y u(x,y)|_{y=0} = \delta_y u(x,y)|_{y=1} = 0$.

Finally we pass the weak form and the boundary conditions to `solve()`, which stores our solutions in `u_1`.

```
solve(a == L, u_1, solver_parameters={'ksp_type': 'cg', 'pc_type': 'none'}, bcs=[bc_x0,bc_x1])
```

We do not need to explicitly define the FEM expansion, or consider matrix assembly and local/ reference co-ordinates, that is, Step 2 and Step 3 - this is implemented by Firedrake.

Method 2

First, we define our solution space `u_2` -

```
u_2 = Function(V, name='u_2')
```

But instead of using the weak form, we consider the Ritz-Galerkin principle which we previously derived by hand -

$$\delta \int_{\Omega} \frac{1}{2} |\nabla u|^2 - uf \, d\Omega = 0$$

Where

```
Ju = (0.5*inner(grad(u_2),grad(u_2)) - u_2*f)*dx
```

Defines the integral, and

```
F = derivative(Ju, u_2, du=v)
```

Differentiates the integral with respect to `u_2` in the direction `v`, producing the weak form. We then pass the our weak form `F = 0` (the integral seen above) to solve and store the solutions in `u_2`.

```
solve(F == 0, u_2, bcs=[bc_x0, bc_x1])
```

Note there is no need to restate the boundary conditions and test and trial functions, as they are used from Method 1.