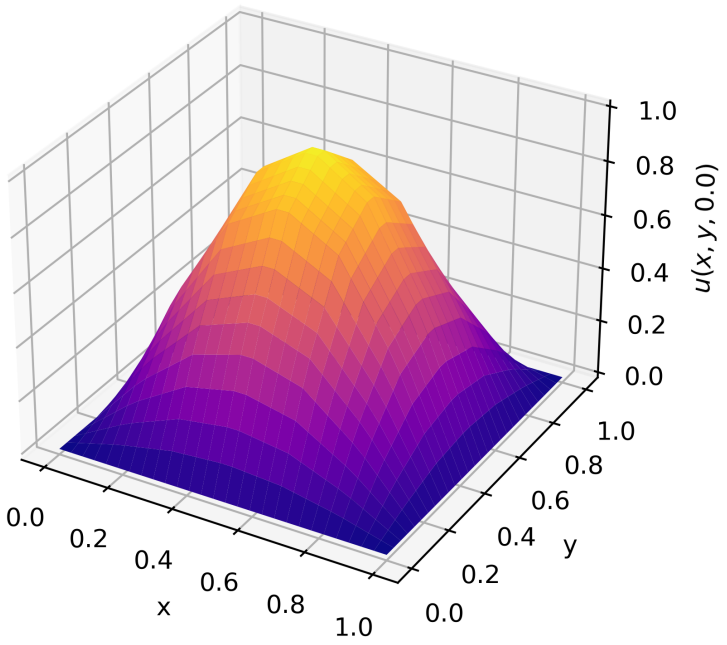
The Finite Element Method

applied to Selected Differential Equations

by Patryk Drozd and Adriana Voloshyna



Computational Physics Project
Maynooth University
2024

Contents

1 Background

There is no doubt that differential equations are ubiquitous in all of physics, and yet often times they can be difficult to solve analytically. Scientists and engineers therefore turn to computational methods in hopes of finding solutions to the differential equations which govern the most fundamental laws of physics. For example, one such set of differential equations are the Navier-Stokes equations, which remain the most reliable tool in understanding the motion of fluids. [1] Despite the widespread practicality of these equations, there is still much to be studied about them. In fact it has not been proven that these equations always have smooth (i.e. infinitely differentiable) solutions in three dimensions. [2] This conundrum is of great interest to the mathematical community, and is one of the seven Millennium Prize Problems, meaning that the Clay Mathematics Institute has offered a prize of one million US dollars for the first correct proof of existence and smoothness or a counterexample. Inspired by the mystery of these equations, we set out to investigate this intriguing world of differential equations using a particular algorithm known as the Finite Element Method (FEM).

2 Introduction

The aim of this project is to solve a number of ordinary and partial differential equations using FEM. To do this, we must begin by studying the mathematical background of the method. A significant portion of this report will focus on the derivation of the necessary tools needed to justify the credibility of this method. A walk-through of the implementation of the method for specific problems will also be provided, alongside results that we obtain from solving these problems.

2.1 Computational Methods

Before choosing our method of computation, we investigated the strengths and weaknesses of several computational methods, such as the Finite Volume Method, Finite Difference Method, and the Finite Element Method.

The Finite Volume Method involves subdividing a space (or volume) into a finite number of cells, creating a discretised collection of what are known as control volumes. Given a partial differential equation which can be written in divergence form, we can use Gauss' Divergence Theorem to rewrite a volume integral into a surface integral, and calculate the flux at each of the surfaces of the cells. This method relies on the conservation of flux, i.e. that the outward flux in each cell must be equal in magnitude to the inward flux entering the cell. [3]

The Finite Difference Method is the most straightforward way to numerically solve a system of differential equations. The domain is discretised into a finite set nodal points, and the derivatives in question are approximated using finite difference (usually a central finite difference). This just means that we take the definition of a derivative from First Principles, but instead of finding the limit as the differential (the small change in x) goes to zero, we give it a very small but non-zero value. Then, for each nodal point we can find the value of a solution at that point, which together gives a numerical solution to a given system of differential equations. This method is efficient in computing solutions on a rectangular domain, but it is difficult to implement on an irregularly shaped domain. [4]

The Finite Element Method also begins with discretising a domain into a finite number of elements. These elements are geometric shapes which together create a mesh for the given domain. To obtain a solution, a set of differential equations together with boundary conditions must first be expressed in what is known as the weak formulation. Then, for a given set of basis functions, we can find a solution for each element of the mesh and together this gives a solution to the system. This method is more mathematically involved than the others and has the least limitations on what type of equations or domains it can be used for.[5] We thus decided that this method would not only be best suited to finding solutions to various types of differential equations, but also would be interesting to study from a more mathematical perspective.

2.2 Background of the Finite Element Method

The Finite Element Method is a popular numerical method of solving differential equations. It can accurately approximate a solution to a boundary value problem when analytic solutions are difficult or impossible to find. Using this method we can model and study physical phenomena such as heat transfer, structural behaviour, fluid flow and electromagnetic potential. Finite Element Analysis is often used in engineering as it can accurately replicate simulations of different types of conditions in a cost effective, safe and efficient way. A very early example of this was NASTRAN, an open source Finite Element Analysis program developed for NASA in the 1960s to aid engineers in structural analysis. The program was so successful that it is still used in aerospace, maritime and automotive industries across the globe today. [6]

3 One Dimensional Poisson Problem

[7][8] To gain an understanding of the algorithm behind the Finite Element Method, we begin by solving the one dimensional Poisson problem. This problem is an easy differential equation to begin our investigation with, and has many physical applications, such as in finding magnetic or electric potential due to charge or current distributions.
Thus we want to find solutions u which satisfy the differential equation

$$-\nabla^2 u = f \qquad \text{on domain } \Omega$$

with the boundary condition

$$u = 0 \qquad \text{on } \partial\Omega$$

Note that in the one dimensional case, the Laplacian ∇^2 is simply the second derivative of u. Taking the domain to be the interval [0,1], the differential equation reduces to

$$-u'' = f \qquad \text{on } [0,1]$$

and the boundary condition can be written as

$$u(0) = u(1) = 0.$$

The above is what is known as the strong formulation of the Poisson problem. We can express this alternatively using the weak formulation.

3.1 Derivation of the Weak Formulation

The weak formulation of the one dimensional Poisson problem can be described as follows:

Given a function space

$Vvcentcolon = \{v|\ v$ is continuous on [0,1], v' is piecewise continuous and bounded on $[0,1], v(0) = v(1) = 0\}$

we must find a $u\ \epsilon\ V$ such that

$$(u', \phi') = (f, \phi) \qquad \forall\, \phi\, \epsilon\, V$$

where $(u, v) := \int_0^1 u(x)v(x)\,dx$ is the scalar product of functions u and v on [0,1].

To see that these formulations are equivalent, observe that if $-u'' = f$, we can multiply both sides by a test function ϕ and integrate over our domain to obtain

$$-\int_0^1 u''(x)\phi(x)\,dx = \int_0^1 f(x)\phi(x)\,dx \qquad \forall\, \phi\, \epsilon\, V$$

after which we can use integration by parts to rewrite this as

$$-([u'(x)\phi(x)]_0^1 + \int_0^1 u'(x)\phi'(x)\,dx) =$$

$$\int_0^1 u'(x)\phi'(x)\,dx - [u'(1)\phi(1) - u'(0)\phi(0)] = \int_0^1 f(x)\phi(x)\,dx \qquad \forall\,\phi\,\epsilon\,V.$$

But since $\phi\,\epsilon\,V$, it must satisfy the boundary conditions $\phi(0) = \phi(1) = 0$, which causes the above to reduce to

$$\int_0^1 u'(x)\phi'(x)\,dx = \int_0^1 f(x)\phi(x)\,dx \qquad \forall\,\phi\,\epsilon\,V \qquad \text{as required.}$$

To show implication in the other direction, and therefore proving equivalence, we repeat the steps above in the reverse direction. Let

$$(u', \phi') = (f, \phi) \qquad \forall\,\phi\,\epsilon\,V$$

and take away zero from the left side of the equation by using the boundary conditions which ϕ must satisfy

$$(u', \phi') - [u'(1)\phi(1) - u'(0)\phi(0)] = (f, \phi) \qquad \forall\,\phi\,\epsilon\,V.$$

We observe that the above can be reduced to

$$(-u'', \phi) = (f, \phi)$$

which can be rewritten as

$$(-u'', \phi) - (f, \phi) = (u'' - f, \phi) = 0$$

Since this must hold $\forall\,\phi\,\epsilon\,V$, we can conclude that

$$-u'' - f = 0 \qquad \text{on } [0, 1]$$

and so we have that the strong and weak form are equivalent.

3.2 Discretisation of the the Function Space

Since computers of our age are not comfortable with infinities, we must discretise the infinite dimensional function space V, into a finite dimensional subspace V_h. To do so, we must create a mesh on which we will use basis functions to approximate a solution to our problem. This mesh provides a way of dividing our continuous domain, in this case the interval [0,1], into a finite amount of nodal points (or vertices) x_i with $x_0 = 0$,..., $x_n = 1$.

On the interval between each nodal point (x_i, x_{i+1}), we can define a **Finite Element** (K, P, Σ) where
- K is a cell of the mesh, also called an element
- P are polynomials on K and
- Σ is a set of degrees of freedom.

Note that for this problem we will be working only with $\Sigma = 1$, as our basis functions will be linear. Later on we will see an example of quadratic basis functions, which have $\Sigma = 2$, but for the purpose of solving the one dimensional Poisson problem, this would add an unnecessary degree of complexity.

Now we have the tools to construct our finite dimensional subspace,

$$V_h := \{v|\ v \text{ is continuous on [0,1], } v|_{K_i}\,\epsilon\,P \text{ for each } i,\ v(0) = v(1) = 0\}.$$

3.3 Basis Functions

The basis functions ϕ_i are simple polynomials which can be used to describe an element of our finite dimensional subspace V_h. A function v has a unique representation in V_h given by $v_h(x) = \sum_{i=0}^{n} v_i \phi_i(x)$ where n is the amount of nodal points in our interval, as before. Each basis function will be scaled by a constant v_i, and must satisfy

$$\phi_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases}$$

Say for example we want to construct a function $v_h(x) = 1 \cdot \phi_{0.25}(x) + 1 \cdot \phi_{0.5}(x) + 1 \cdot \phi_{0.75}(x)$. Figure 1 on the left shows how the sum of the individual (scaled) bases superimpose to form our function. Similarly we can scale the basis functions to construct a function $v_h(x) = 1 \cdot \phi_{0.25}(x) + 3 \cdot \phi_{0.5}(x) + 2 \cdot \phi_{0.75}(x)$, as can be seen in figure 2.

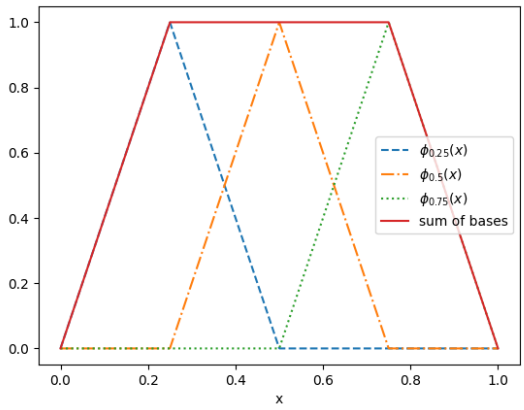


Figure 1: Sum of linear bases

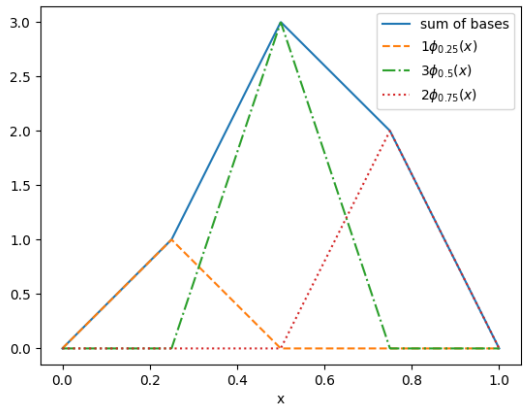


Figure 2: Sum of scaled linear bases

3.4 Discretisation of the Weak Formulation

The weak form of the Poisson problem can now be discretised to the following: For a function space V_h (as described previously), we must find a $u_h \, \epsilon \, V_h$ such that

$$(u_h', \phi_h') = (f, \phi_h) \qquad \forall \, \phi_h \, \epsilon \, V_h.$$

We can write u_h' in terms of its basis functions to obtain:

$$((\sum_{j=0}^{n} u_j \phi_j)', \phi_h') = (f, \phi_h) \qquad \forall \, \phi_h \, \epsilon \, V_h.$$

Furthermore, since any function ϕ_h can be written as a linear composition of basis functions, it is enough to show that

$$((\sum_{j=0}^{n} u_j \phi_j)', \phi_i') = (f, \phi_i) \qquad \forall \, 0 \leq i \leq n$$

$$\iff \sum_{j=0}^{n} u_j (\phi_j', \phi_i') = (f, \phi_i) \qquad \forall \, 0 \leq i \leq n.$$

Now we search for a sequence of constants (u_j) which satisfy the equivalence above.

3.5 Linear System Representation

To do this computationally, we can represent the linear system in the form of a matrix equation

$$\begin{pmatrix} (\phi_1', \phi_1') & (\phi_1', \phi_2') & \dots & (\phi_1', \phi_n') \\ (\phi_2', \phi_1') & (\phi_2', \phi_2') & \dots & (\phi_2', \phi_n') \\ \vdots & \vdots & \ddots & \vdots \\ (\phi_n', \phi_1') & (\phi_n', \phi_2') & \dots & (\phi_n', \phi_n') \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} (f, \phi_1) \\ (f, \phi_2) \\ \vdots \\ (f, \phi_n) \end{pmatrix}$$

to find a vector $\vec{u} \, \epsilon \, \mathbb{R}^n$ which satisfies this equation. For clarity, let us rewrite the above equation as

$$A\vec{u} = F$$

Solving a linear system like this is not difficult - we simply multiply both sides of the equation with the inverse of A to obtain what $\vec{u}$ is equal to. All that remains is to find the components of A, i.e. the scalar product of the basis functions.

3.6 Matrix A

Due to the symmetry of the matrix, we begin by solving the diagonal elements, which are all of the form (ϕ_i', ϕ_i'). Lets take a look at one basis function ϕ_i which is non-zero only on the interval $[x_{i-1}$, $x_{i+1}]$. If h is the distance from x_{i-1} to x_i and the height of the basis function is set to 1, then

$$\phi_i(x) = \frac{x}{h} \qquad \text{when restricted to the interval} \qquad [x_{i-1}, x_i].$$

By symmetry,

$$\phi_i(x) = 1 - \frac{x}{h} \qquad \text{on the interval} \qquad [x_i, x_{i+1}].$$

Then

$$\int_0^1 \phi_i'(x)\phi_i'(x)\,dx = \int_{x_{i-1}}^{x_{i+1}} \phi_i'(x)\phi_i'(x)\,dx = \int_{x_{i-1}}^{x_i} \left(\frac{1}{h}\right)\left(\frac{1}{h}\right)\,dx + \int_{x_i}^{x_{i+1}} \left(-\frac{1}{h}\right)\left(-\frac{1}{h}\right)\,dx$$

$$= \left[\frac{x}{h^2}\right]_{x_{i-1}}^{x_i} + \left[\frac{x}{h^2}\right]_{x_i}^{x_{i+1}} = \frac{h}{h^2} + \frac{h}{h^2} = \frac{2}{h}.$$

Now to find the elements next to the diagonal, we must calculate $(\phi_i', \phi_{i+1}') = (\phi_{i+1}', \phi_i')$. Since $\phi_i = 0$ anywhere outside the interval $[x_{i-1}$, $x_{i+1}]$, we can limit our integral to the same interval. Moreover, $\phi_{i+1} = 0$ on the interval $[x_{i-1}$, $x_i]$, so we can further restrict our limits of integration to x_i and x_{i+1}.

$$\int_{x_i}^{x_{i+1}} \phi_i'(x)\phi_{i+1}'(x)\,dx = \int_{x_i}^{x_{i+1}} \left(-\frac{1}{h}\right)\left(\frac{1}{h}\right)\,dx = \left[-\frac{x}{h^2}\right]_{x_i}^{x_{i+1}} = -\frac{1}{h}.$$

Finally observe that any matrix elements other than those which we have calculated must equal to zero, as the two basis functions in the scalar product will never be non-zero on the same interval. This matrix is also clearly symmetric. This comes from the fact that basis elements commute, i.e. $\phi_i\phi_j = \phi_j\phi_i$.

3.7 Matrix F

To begin solving the left hand side of the equation, we must chose a function f. For the sake of simplicity, let $f = 1$. Then, for any basis function $\phi_i(x)$,

$$(f, \phi_i) = \int_0^1 f(x)\phi_i(x)\,dx = (1)\int_{x_{i-1}}^{x_{i+1}} \phi_i(x)\,dx = \int_{x_{i-1}}^{x_i} \left(\frac{x}{h}\right)\,dx + \int_{x_i}^{x_{i+1}} \left(1 - \frac{x}{h}\right)\,dx = h.$$

3.8 Numerical Implementation and Solution

Although it is possible to solve the scalar products of basis functions analytically, it can become tedious for large mesh sizes and non linear basis functions. Using the trapezoidal integration function np.trapz, we can solve for the elements of the matrix numerically. To see that this numerical method can reproduce the same results that we achieved analytically, we plotted the elements of the matrix A. Observe in figures 5 and 6 the results for a system with 9 nodal points and a system with 50 nodal points. Indeed we get a tridiagonal symmetric matrix with non-zero elements only along the diagonal and the upper and lower diagonal.

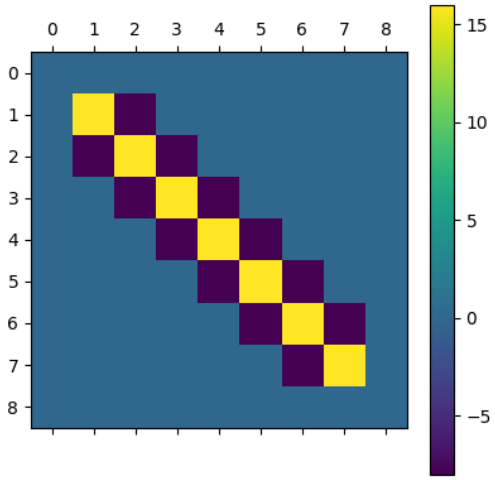


Figure 3: Matrix A for 9 nodal points

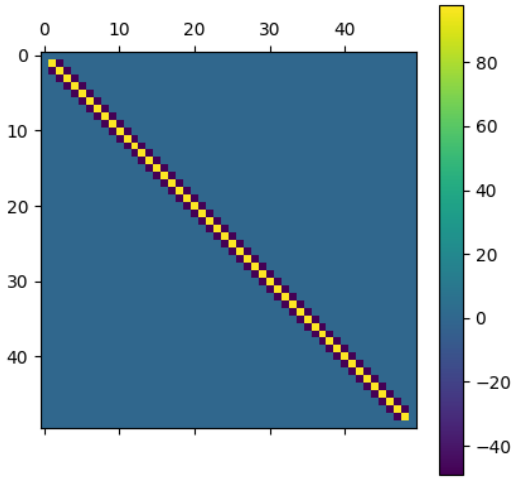


Figure 4: Matrix A for 50 nodal points

Now that we have a linear system $A\vec{u} = F$, and the tools to obtain this equation numerically, we can look for solutions $\vec{u}$. To do this, it is necessary to create three nested for loops. For each element of our mesh, we find the scalar product of our function f with our basis functions ϕ_i for all i. Then, for each basis function ϕ_i, we take the scalar product of ϕ_i and ϕ_j, for all j, which gives us a row of the matrix A for each iteration of i. Using numpy.linalg.pinv we find the inverse of this matrix, and thus solve for $\vec{u}$. We can then plot $\vec{u}$ to see the solutions to the one dimensional Poisson problem. These plots can be seen in in figures 5 and 6. Note how the curve is smoother when we consider a finer mesh with more nodal points.

We also consider $f = x$, and obtain a solution for $\vec{u}$ which can be seen in figure 7.

To check that our solutions are correct, we compare our numerical results to the analytic solutions of these equations. Using our knowledge of second-order linear ordinary differential equations, we obtain the analytic solutions

$$u(x) = \frac{1}{2}x(x-1) \qquad \text{for} \quad f(x) = 1,$$

$$u(x) = \frac{1}{6}x(x^2-1) \qquad \text{for} \quad f(x) = x.$$

If we plot these functions we can see that for a sufficient number of nodal points n our numerical solutions are an accurate approximation of the analytic solutions.

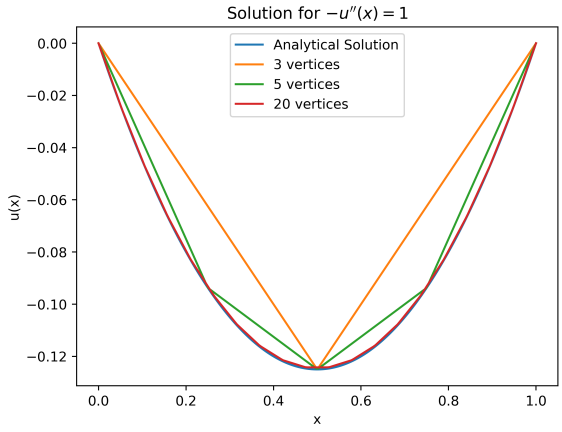


Figure 5: Solutions for various mesh sizes

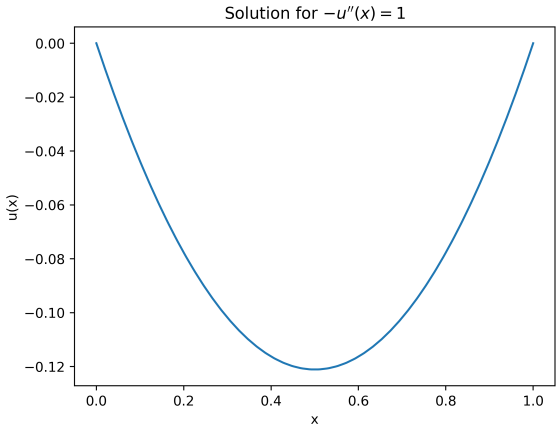


Figure 6: Solution for 50 nodal points

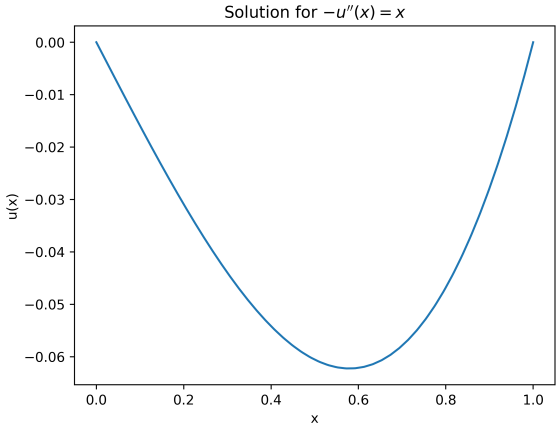


Figure 7: Solution for 50 nodal points

4 Two Dimensional Poisson Problem

To solve the two dimensional Poisson problem [9], we wish to find a function $u(x, y)$ which satisfies

$$\nabla^2 u = f \text{ on } \Omega \quad \text{with mixed Neumann and Dirichlet boundary conditions.}$$

Here the domain is a unit length square $[0, 1] \times [0, 1]$, and the Laplacian $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. We decided to experiment with the types of boundary conditions which we can impose. The motivation behind this was the idea that often in physics we have a fluid which flows in from one end of a "box", i.e. we enforce that our solution has a non zero derivative on one side of our domain, and it dissipates on the remaining boundaries of the domain. We therefore impose Neumann boundary conditions (boundary conditions on the directional derivative of u as opposed to u itself) of the form

$$\nabla u(x, y) \cdot \hat{n} = -x(x - 1) \qquad \forall\, (x, y)\, \epsilon\, \partial\Omega_N \subset \partial\Omega,$$

and the Dirichlet boundary conditions are $u(x, y) = 0$ on $\partial\Omega_D \subset \partial\Omega$ and we choose

$$\partial\Omega_N = \{1\} \times [0, 1] \quad \text{and} \quad \partial\Omega_D = (\{0\} \times [0, 1]) \cup ([0, 1] \times \{0\}) \cup ([0, 1] \times \{1\}).$$

The two dimensional Poisson problem can also be written in its weak formulation. To do this, we must first define an infinite dimensional function space:

$$V := \{v(x,y) |\, v, \nabla v \,\epsilon\, L^2(\Omega), v = 0 \text{ on } \partial\Omega_D\}.$$

Here $L^2(\Omega)$ is a space of functions which are square integrable on the given domain. [10] The reason we need to introduce a slightly complicated function space, is to ensure uniqueness of a solution. When we proceed to search for solutions on our mesh, we could encounter problems of uniqueness of a function at the boundary of each cell. But since a line has Lebesgue Measure zero (meaning that a line in the real plane does not contribute to the value of an integral), we can surpass this problem by defining functions to be in a sense equal if they have the same "size", i.e. they are equal in their L^2 norm. In this test space, we only restrict our functions with the Dirichlet boundary condition and not the Neumann boundary condition. The latter will appear in the weak formulation of the problem.

4.1 The Weak Formulation

Now we can find the weak formulation of the Poisson problem in the same way as we did previously. [11] We multiply both sides of the equation by a test function $\phi(x,y)$ and integrate both sides over the entire domain. Then using integration by parts (in particular an application of Green's first identity) and applying the Dirichlet boundary condition we obtain

$$\int_\Omega \nabla^2 u\, \phi \, dxdy = -\int_\Omega \nabla u \cdot \nabla \phi \, dxdy + \int_{\partial\Omega} \nabla u \cdot \hat{n}\, \phi \, dxdy = \int_\Omega f(x,y)\phi(x,y)\ dxdy \qquad \forall\, \phi \,\epsilon\, V.$$

Implementing the Neumann boundary conditions gives

$$\int_\Omega \nabla u \cdot \nabla \phi \, dxdy = -\int_\Omega f\phi \ dxdy - \int_{\partial\Omega_N} x(x-1)\, \phi \, dxdy \qquad \forall\, \phi \,\epsilon\, V.$$

4.2 Basis Functions and Discretisation

We once again only consider a discretised subspace of V, namely V_h where we restrict each $v_h \,\epsilon\, V_h$ to be polynomial on each element of our mesh. We similarly define basis functions ϕ_i to describe the functions in our discretised subspace, except this time the basis functions must describe two dimensional functions. This is done by subdividing the square $[0,1]\times[0,1]$ into a grid of $n \times n$ smaller squares. On each such square, we want a basis function to be 1 on one corner of the square and 0 on the other three. This can be achieved by taking the product of a basis function on the x-axis with a similar basis function on the y-axis, i.e.

$$\phi_{ab}(x,y) = \phi_a(x) \cdot \phi_b(y).$$

But we also want the neighbouring three squares to peak at the same point, i.e. we want to create an almost pyramidal shape across four adjacent squares. Letting h be the length of each interval (and therefore also the side length of each square) we can define

$$\phi_i(x,y) := \begin{cases} \phi_{00}(x,y) = \frac{xy}{h^2} & (x,y)\,\epsilon\,[x_{i-1},x_i]\times[y_{i-1},y_i] \\ \phi_{01}(x,y) = \frac{x}{h}(1-\frac{y}{h}) & (x,y)\,\epsilon\,[x_{i-1},x_i]\times[y_i,y_{i+1}] \\ \phi_{10}(x,y) = (1-\frac{x}{h})\frac{y}{h} & (x,y)\,\epsilon\,[x_i,x_{i+1}]\times[y_{i-1},y_i] \\ \phi_{11}(x,y) = (1-\frac{x}{h})(1-\frac{y}{h}) & (x,y)\,\epsilon\,[x_i,x_{i+1}]\times[y_i,y_{i+1}] \\ 0 & \text{elsewhere} \end{cases}$$

where i denotes where the peak of this basis function will be. Note that this description of the basis function is not complete. In the code, we define the basis functions in terms of i, so that we obtain

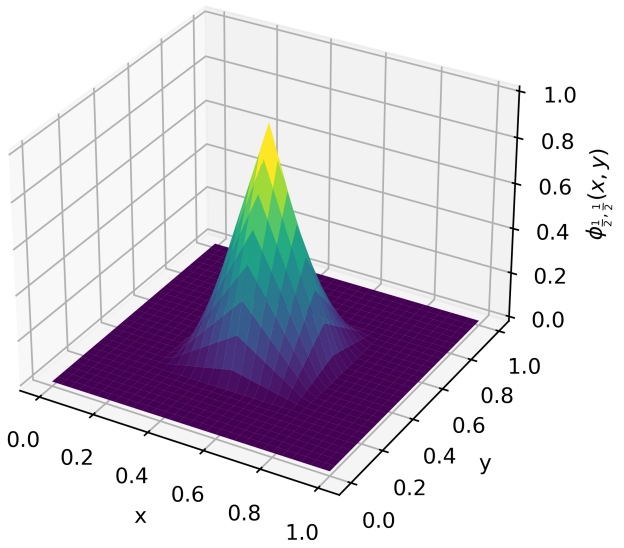


Figure 8: Two dimensional basis function with peak at $(x, y) = (0.5, 0.5)$ and distance between nodes $h = 0.25$

a general representation of basis functions for any element. Here however we have the specific basis function which peaks only at a chosen i. The basis functions are linear on x and y, and still polynomial on each element of our mesh. Observe in figure 8 what one such basis function looks like.

We now begin to assemble our problem as we did before into a linear matrix equation. The function $u(x, y)$ can be written as $u_h(x, y) = \sum_{i=0}^{n} u_i \phi_i(x, y)$, and similarly the test function ϕ can be replaced with a basis function $\phi_i(x, y)$ and summed over all $0 \leq i \leq n$. The problem is expressed in the form

$$A\vec{u} = F,$$

where $\vec{u}$ is the same as for the one dimensional case, just a sequence of constants which will be our solution. Matrix A (of size $n \times n$) will have elements of the form

$$\int_{\Omega} \nabla\phi_i \cdot \nabla\phi_j \, dxdy.$$

Note also that while the gradient of a scalar would give us a vector, we immediately dot product it with another vector (giving us back a scalar), so no complications arise in terms of having to do vector integration. The integration part of the problem we done simply using two np.trapz functions.

Matrix F (of size $n \times 1$) has elements of the form

$$-\int_{\Omega} f(x, y)\phi_j \; dxdy - x(x-1)\int_{\partial\Omega_N} \phi_j \, dy.$$

Recall that the second integral above is calculated only on the line $\{1\} \times [0, 1]$, hence why we can treat it as a line integral in the y direction.

4.3 Implementation and Results

The implementation procedure is similar to what was done for the 1 dimensional case. It is interesting to note that in most popular libraries which provide easy interface to use FEM, the meshes have somewhat randomly placed nodes and triangular elements. For the 1 dimensional and 2 dimensional problems, we decided to use square elements because of the ease of implementation. Triangular elements and their associated nodes would have a different set of basis functions and the integrals needed for the weak formulation of the equations would need to be evaluated over triangular domains. On the other hand square meshes only need 2 arrays to be defined. Each basis function will only need to be shifted according to the position of the node and the integrals will be very easy to visualise as they are on square domains. It also means that the implementation for the 2 dimensional case can be relatively easily adapted from the 1 dimensional case.

Another key difference in implementation with respect to the one dimensional case is the way in which the matrix elements are defined. When iterating over the domain Ω, we begin by looking at the basis function in the top left corner of the domain, and continue downwards along the leftmost "strip" of the domain. We then continue listing the basis functions on the second "strip" starting from the top, and so forth until we reach the rightmost strip of the domain, where the Neumann boundary $\partial\Omega_N$ lies. However note that every time we reach the end of a strip, our basis functions must vanish due to the Dirichlet boundary conditions, and the same goes for the start of the next strip. In this example, we chose $n = 7$ to discretise our domain, meaning we obtain a 7×7 grid. On the first vertical strip five nodes out of seven are non zero, which results in the segmented diagonal line we see in the matrix in figure 9. Note also how the bottom right corner of the matrix has a lower value than the rest of the diagonal - this is due to the fact that the last basis function along the boundary will not be a full "pyramidal" shape, but rather half of one. As expected, the matrix we obtain for A is still symmetric, with a segmented main diagonal and similar upper and lower diagonal as what we obtained for the one dimensional case.

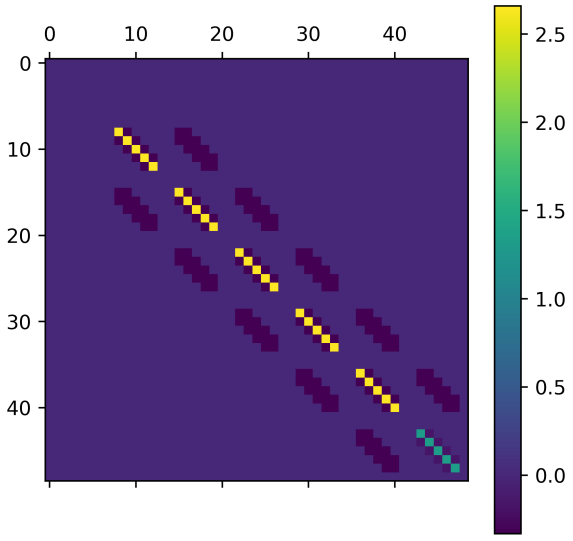


Figure 9: Matrix A for 49 nodal points (with Dirichlet and Neumann BCs)

Figure 10 shows the two dimensional plot of the solution u to this problem with the forcing function

$$f(x, y) = -2y.$$

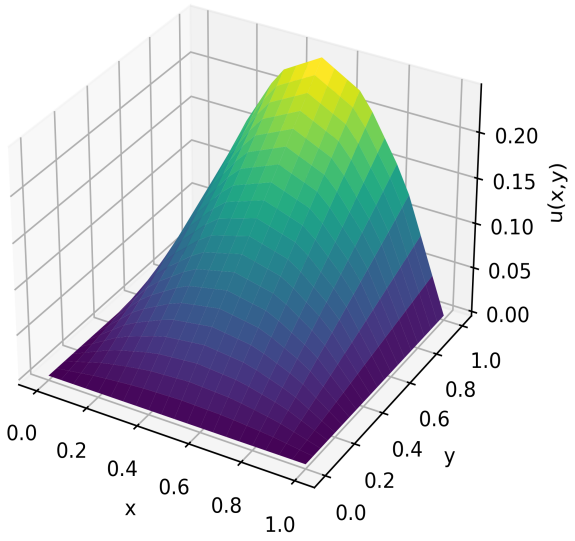


Figure 10: Solution to $\nabla^2 u = -2y$ with mixed boundary conditions

To verify that our numerical solution is correct, we would have to know the analytic solution to our PDE. Fortunately, the choice of this problem was derived from a chosen solution i.e. for the function

$$u(x, y) = -yx(x - 1)$$

we find a suitable $f(x, y)$ so that the equation $\nabla^2 u(x, y) = f(x, y)$ holds. Since

$$\nabla^2(-yx(x - 1)) = -2y,$$

we obtain our required forcing function $f(x, y) = -2y$. The solution u also contains the information for the boundary conditions required for the problem. Plotting the predetermined analytic solution, we are able to see that the solutions we obtained numerically match their precise mathematical form.

4.4 Another Two Dimensional Poisson Problem

Now that we have the necessary computational tools, we also decided to solve the two dimensional Poisson problem for the forcing function

$$f(x, y) = -2\pi^2 \sin(\pi x) \sin(\pi y)$$

with only the Dirichlet boundary condition

$$u(x, y) = 0 \,\text{on}\ \partial\Omega.$$

The results we obtain for the matrix A and the solution $u(x, y)$ are seen in figures 11 and 12.
Using the same principle, we know that the analytic solution of this problem is

$$u(x, y) = -sin(\pi x)sin(\pi y).$$

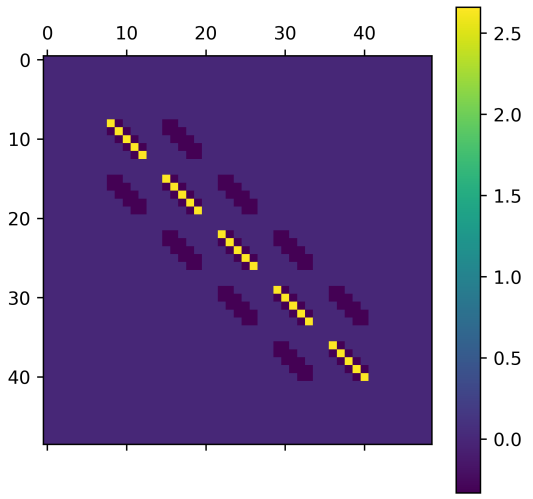


Figure 11: Matrix A for 49 nodal points

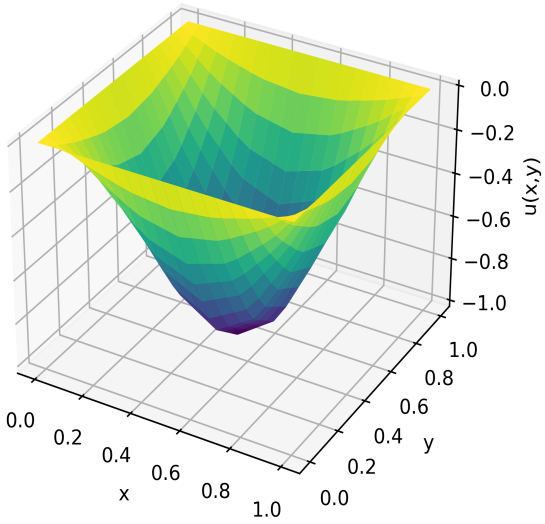


Figure 12: Solution for 49 nodal points

4.5 A note on Error Analysis

As with any computational method, it is crucial that we study the errors that arise with this method. [12] One of the main errors we encounter is the discretisation error. As we can see in the 1 dimensional Poisson problem figure 5, the mesh size we chose plays an important role in the accuracy of our solutions. Throughout the project we often times first calculate solutions on a courser mesh grid, and when we obtain something that we are happy with, we refine the mesh to a finer grid.

There is also an error associated with the estimation of functions on each cell of our mesh, also known as the interpolation error. In the beginning we use linear polynomials as our basis functions to discretise our functions, but later we expand our investigation to higher order polynomials. We found that for a sufficiently large number of nodes, the linear basis functions produced quite accurate results for what we were solving, however if we were to try more advanced problems on more advanced domains, we might have to turn to higher order polynomials and denser meshes for better accuracy.

Then there is the usual error associated with the other numerical methods used in this project such as trapezoidal integration, finite difference method of obtaining derivatives etc. We found that it is important to increase the accuracy of the numerical integration as you increase the number of nodes.

To measure the error of our numerically derived solutions, we use the L^2 norm which was mentioned earlier. This involves measuring the difference of the square integral of our known analytic solution and the numerical solution which we derive. For example in the second two dimensional Poisson problem (Section 4.4) we found that for a larger number of nodes, the L^2 error was smaller. We choose our integral step size to be 1000 per element, and obtain the following results. On a 5×5 grid, the error of our solution was of magnitude ≈ 0.030655. However if we increase the size of the grid to 7×7, the error decreased to ≈ 0.014620.

5 Heat Equation

The heat equation is a fundamental partial differential equation in both pure and applied mathematics. It describes the diffusion of a property such as heat in a given domain over some time t. Here we aim

to solve the following form of the heat equation:

$$\frac{\partial u}{\partial t} = \nabla^2 u + f$$

where $u = u(x, y, t)$ is a function of two spacial variables and one time variable, and $f(x, y)$ is some forcing function. [13] For this problem we let the spacial domain Ω be the unit square $[0, 1] \times [0, 1]$ and impose the usual Dirichlet boundary conditions

$$u(x, y, t) = 0 \quad \text{on } \partial\Omega.$$

We additionally impose initial conditions

$$u(x, y, 0) = \sin(\pi x) \sin(\pi y).$$

5.1 Discretisation of Time

To solve the time dependent PDE, we must first discretise the time derivative, and then obtain a stationary problem at each timestep. If we let s be a particular time step then the time derivative can be discretised using finite (backward) difference as

$$\left(\frac{\partial u}{\partial t}\right)^{s+1} \approx \frac{u^{s+1} - u^s}{\Delta t}$$

where superscript s denotes the value of the function at the timestep s and Δt is a small but nonzero difference in time. We now obtain an expression for the problem at a timestep s

$$\frac{u^{s+1} - u^s}{\Delta t} = \nabla^2 u^{s+1} + f.$$

Since f is independent of time, we exclude the superscript timestep for f. But we know what u^0 is, as this is the initial condition which we specified earlier, so we can define a sequence of equations for u^{s+1} (given that we have u^s) as follows

$$u^0 = \sin(\pi x) \sin(\pi y)$$

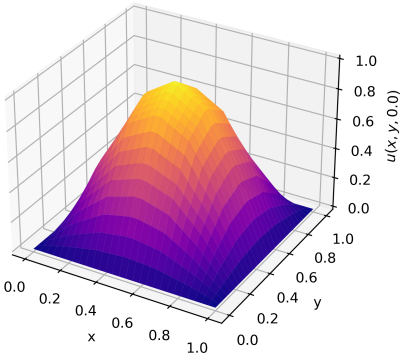
$$u^{s+1} - \nabla^2 u^{s+1} \Delta t = u^s + f\Delta t \qquad \text{for } s = 0, 1, 2, \ldots$$

5.2 The Weak Formulation and Spacial Discretisation

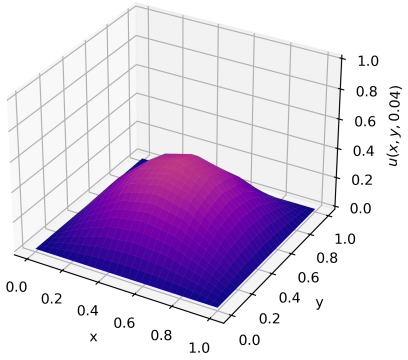
The procedure for obtaining the weak formulation for each time step is the same as usual. We define an appropriate function space for this problem. We then multiply by a test function ϕ on both sides of the equation and integrate over the entire domain. Then we rewrite the integrals with the Laplacian term using integration by parts in order to reduce the order of the differential equation. Lastly we apply the Dirichlet boundary conditions and we obtain the following:

$$\int_\Omega u^{s+1}\phi \, dxdy + \int_\Omega \Delta t \nabla u^{s+1} \cdot \nabla\phi \, dxdy = \int_\Omega u^s \phi \, dxdy + \int_\Omega \Delta t f \phi \, dxdy.$$

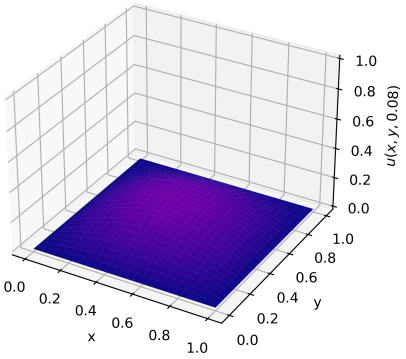
We discretise this formulation using the same basis functions as in the two dimensional Poisson problem. Then for each given u^s, starting with u^0, we solve the weak form by writing it as a linear matrix equation to obtain a solution for u^{s+1}.



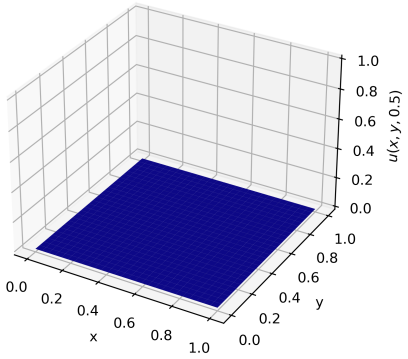
(a) Timestep 0



(b) Timestep 4



(c) Timestep 8



(d) Timestep 50

Figure 13: Snapshots of the solution to $\frac{\partial u}{\partial t} = \nabla^2 u$ with Dirichlet boundary conditions and initial conditions $u(x, y, 0) = \sin(\pi x) \sin(\pi y)$

5.3 Implementation and Results

The method for solving this problem is similar to what was done in the two dimensional Poisson problem. The main difference in the implementation is that we iterate with a time step Δt. This means that we create an additional for loop, and obtain a sequence of solutions rather than just one solution. Choosing $f(x, y) = 0$ we obtain a specific sequence of solutions to the problem. To show the solution as it evolves over time, we created a short animation, which you can see here. Additionally, in figure 13, observe selected snapshots of the evolution of $u(x, y, t)$ over 50 timesteps with $\Delta t = 0.01$. This figure showcases the typical behaviour of heat dissipation, where the heat flows from areas of high concentration into the surrounding areas of low concentration. Also, as we physically expect, the heat dissipates quickly at the beginning (when the temperature difference is large), and the dissipation slows down in the later timesteps.

6 Stokes Equations

Lastly, we return to the equations describing fluid flow. [14] The Stokes equations is a simpler case of the notorious Navier-Stokes equations, where the fluid is assumed to be viscous but with low velocity. These equations are often associated with Stokes flow, also known as creeping flow, and can be used to model the behaviour of a slow fluid such as the flow of bodily fluids, paint, water flow in soil etc.[15][16]

Solving these partial differential equations will be different to what we've done so far, as we are dealing with a system of vector equations. Observe below the following formulation of the Stokes problem:

$$\mu\nabla^2\vec{u} - \nabla p = \vec{f}$$
$$\nabla \cdot \vec{u} = 0$$

where

- μ is dynamic viscosity
- $\vec{u}$ is the velocity of the fluid
- p is fluid pressure
- $\vec{f}$ describes external forces (e.g. gravity).

Given a forcing function $\vec{f}$ (and a constant μ) we aim to find a suitable $\vec{u}$ and p which satisfy the equations above. We let our domain Ω the unit square $[0,1]\times[0,1]$, and impose the following Dirichlet boundary conditions on $\vec{u}$ and p:

$$\vec{u} = \vec{0} \quad \text{on } \partial\Omega$$
$$p(x,y) = 0 \quad \text{on } \partial\Omega.$$

Let $\vec{u} = (u_1, u_2)$ where u_1 is the horizontal velocity of the fluid and u_2 is the vertical velocity. Similarly, let $\vec{f} = (f_1, f_2)$ We can then separate the first Stokes equation component-wise to obtain the following system of equations:

$$\mu\nabla^2 u_1 - \frac{\partial p}{\partial x} = f_1$$
$$\mu\nabla^2 u_2 - \frac{\partial p}{\partial y} = f_2$$
$$\nabla \cdot \vec{u} = 0$$

These equations can each be put into its weak formulation, and then solved simultaneously in one matrix equation using a clever arrangement of the system.

6.1 The Weak Formulation

We obtain the weak formulation using the same method as usual. Given three test functions ϕ_1, ϕ_2, ϕ_3 the weak formulations of each equation can be expressed as:

$$\mu\int_\Omega \nabla u_1 \cdot \nabla\phi_1 \, dxdy - \int_\Omega p\frac{\partial\phi_1}{\partial x}\, dxdy = \int_\Omega f_1\phi_1 \, dxdy$$
$$\mu\int_\Omega \nabla u_2 \cdot \nabla\phi_2 \, dxdy - \int_\Omega p\frac{\partial\phi_2}{\partial y}\, dxdy = \int_\Omega f_2\phi_2 \, dxdy$$
$$\int_\Omega (\nabla \cdot \vec{u})\phi_3 = 0$$

6.2 Basis Functions

To discretise the weak formulation, we first define a set of basis functions for our mesh. This time we must introduce quadratic basis functions to satisfy what is known as the Ladyzhenskaya–Babuška–Brezzi (LBB) condition. The discretisation of Stokes flow involves certain saddle-point problems, which can lead to instability in the code. In simplified terms, this condition ensures that the discretisation of Stokes flow guarantees stability and convergence. We thus use mixed basis functions, linear and quadratic, to discretise our functions.[17] The combination of bases we use is known as the Taylor-Hood elements. This means we approximate the pressure function p using linear basis elements, and the velocity components u_1, u_2 using quadratic basis elements.

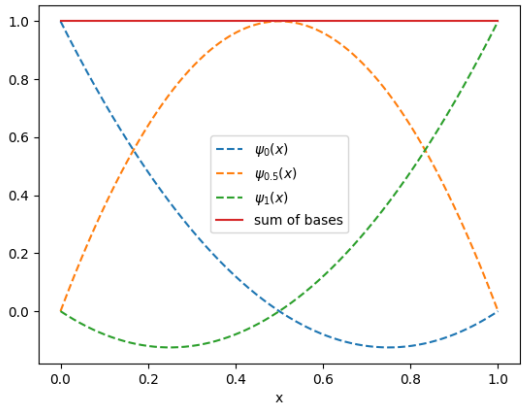


Figure 14: Sum of quadratic bases

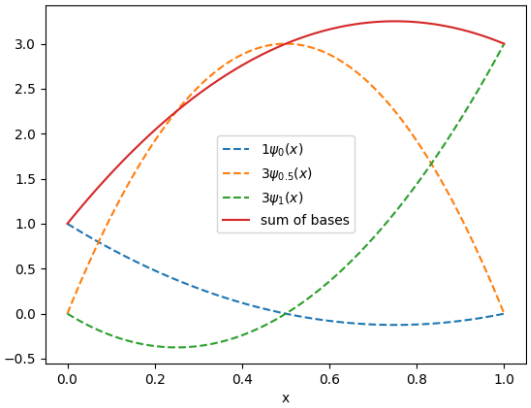


Figure 15: Sum of scaled quadratic bases

To gain and understanding of the two dimensional quadratic basis functions, we begin by looking at the one dimensional case. Here, for each element on our mesh, we use three quadratic functions to approximate a function on this element. We can see the construction of a function $v_h(x) = 1 \cdot \psi_{0.25}(x) + 1 \cdot \psi_{0.5}(x) + 1 \cdot \psi_{0.75}(x)$ in figure 14. We can also scale the basis functions to construct a function $v_h(x) = 1 \cdot \psi_{0.25}(x) + 3 \cdot \psi_{0.5}(x) + 3 \cdot \psi_{0.75}(x)$, as is seen in figure 15.

On each element we therefore have the two nodes at the endpoints, and an additional node in the centre, where the peak of the central quadratic basis function is. However defining these basis functions computationally is not as straightforward as for the linear case.

Let h be the the distance between two consecutive nodes, and let $x = i$ denote the peak of our basis function. On each odd node of our mesh (node 1, node 3, ...) we define a "n-shaped" basis function, a plot of which can be seen in figure 16. We write this as

$$\psi_i^{odd}(x) = \begin{cases} 1 - \frac{(x-i)^2}{h^2} & x \,\epsilon\, [i-h,\, i+h] \\ 0 & \text{elsewhere} \end{cases}$$

On each even node of our mesh, (node 0, node 2, ...) we define an "pointy" basis function, which can be seen in figure 17. This can be expressed as a piecewise function

$$\psi_i^{even}(x) = \begin{cases} \frac{1}{2h^2}(x-(i-2h))(x-(i-h)) & x \,\epsilon\, [i-2h,\, i] \\ \frac{1}{2h^2}(x-(i+2h))(x-(i+h)) & x \,\epsilon\, [i,\, i+2h] \\ 0 & \text{elsewhere} \end{cases}$$

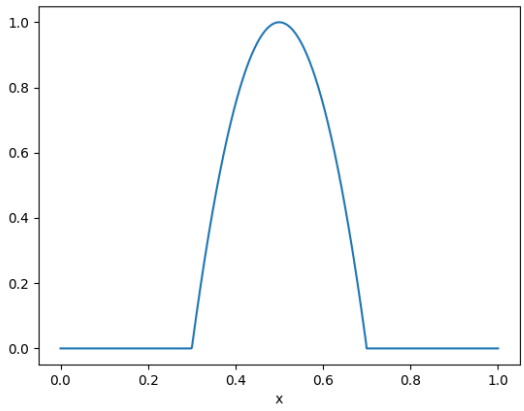


Figure 16: Odd Quadratic Basis Function

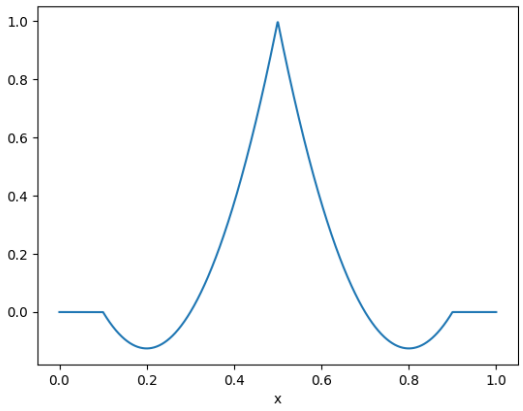


Figure 17: Even Quadratic Basis Function

Here we explicitly define the function in terms of the peak i of each function, as there is a subtle but important difference between the placement of the two types of functions. Note how the even basis function fits exactly into each element of the mesh, but the odd basis function "spills over" to nearby elements. Also observe that the peak is not the same for both functions. If i_{even} is the peak of the even function and i_{odd} is the peak of the odd function, then $i_{even} = i_{odd} \pm h$.

To generalise this notion to two dimensions, we consider a similar process to what we did in the two dimensional linear basis. We take a one dimensional basis function (say at $x = a$, and multiply it by another one dimensional basis function (say at $y = b$), to obtain the two dimensional basis function (at $(x, y) = (a, b)$). But now we have two types of one dimensional basis functions to chose from, so we actually obtain 4 different types of combinations:

- Even $\times$ even
- Odd $\times$ odd
- Odd $\times$ even
- Even $\times$ odd.

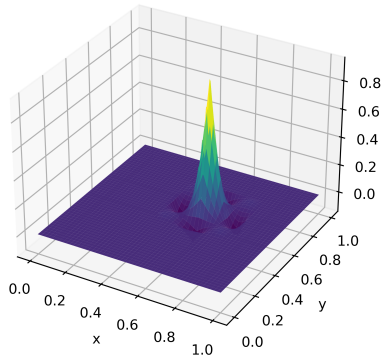
While it is possible to manually calculate all the possible combinations, it is quite laborious and unnecessary for our purposes. We can instead plot each of the types of basis functions, to give us a better intuition of their structure. Observe the results we obtain for the different types of basis functions in figure 18.

6.3 Discretisation of the Weak Formulation

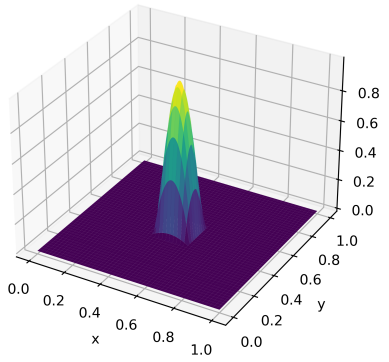
As stated earlier, we discretise the velocity componenets using quadratic basis functions and the pressure using linear basis functions. Let

$$(u_1(x, y))_h = \sum_{i=0}^{n} u_i^1 \psi_i(x, y)$$

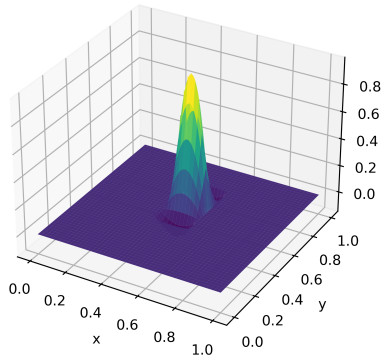
$$(u_2(x, y))_h = \sum_{i=0}^{n} u_i^2 \psi_i(x, y)$$



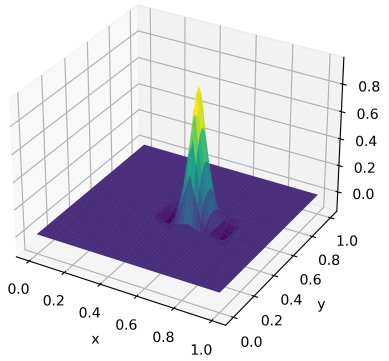
(a) Even × even, (0.6, 0.6)



(b) Odd × odd, (0.5, 0.5)



(c) Odd × even, (0.5, 0.6)



(d) Even × odd, (0.6, 0.5)

Figure 18: Two dimensional quadratic basis function

$$(p(x,y))_h = \sum_{i=0}^{n} p_i \phi_i(x,y).$$

Then we can rewrite the weak formulation in its discretised terms:

$$\sum_{i=0}^{n} u_i^1 \left(\mu \int_\Omega \nabla\psi_i \cdot \nabla\psi_j \, dxdy \right) - \sum_{i=0}^{n} p_i \int_\Omega \phi_i \frac{\partial \psi_j}{\partial x} \, dxdy = \int_\Omega f_1 \psi_j \, dxdy \tag{1}$$

$$\sum_{i=0}^{n} u_i^2 \left(\mu \int_\Omega \nabla\psi_i \cdot \nabla\psi_j \, dxdy \right) - \sum_{i=0}^{n} p_i \int_\Omega \phi_i \frac{\partial \psi_j}{\partial y} \, dxdy = \int_\Omega f_2 \psi_j \, dxdy \tag{2}$$

$$\sum_{i=0}^{n} u_i^1 \int_\Omega \frac{\partial \psi_j}{\partial x} \phi_j \, dxdy + \sum_{i=0}^{n} u_i^2 \int_\Omega \frac{\partial \psi_j}{\partial y} \phi_j \, dxdy = 0 \tag{3}$$

Note the choice of test basis functions by which we multiply the equation. The mixed basis functions are only required in the PDEs which contain both pressure elements and velocity elements, so the first two equations are being multiplied by a quadratic basis function, and the last one is multiplied by a linear basis function.

6.4 Matrix Formulation

We now wish to write the above three equations as a matrix, to solve for a sequence of constants (u_i^1), (u_i^2) and (p_i), and clever trick to compile these into one matrix equation was promised.

In the formulation $A\vec{v} = F$, we begin by finding the matrix A, as usual. This matrix will be divided into nine (3×3) main parts. We first label some of the integral terms in the above equations for clarity.

$$a_{ij} := \mu \int_\Omega \nabla\psi_i \cdot \nabla\psi_j \, dxdy$$

$$b_{ij} := -\int_\Omega \phi_i \frac{\partial\psi_j}{\partial x} \, dxdy$$

$$c_{ij} := -\int_\Omega \phi_i \frac{\partial\psi_j}{\partial y} \, dxdy$$

$$d_j^1 := \int_\Omega f_1\psi_j \, dxdy \qquad\qquad d_j^2 := \int_\Omega f_2\psi_j \, dxdy$$

If we look at the left side of the first equation (1), we can see that there is an a_{ij} term associated with the u_i^1 components and a b_{ij} term associated with the p_i components. This will create the first three (horizontal) blocks of the matrix :

$$\left(\begin{array}{ccc|ccc|ccc} a_{00} & \dots & a_{0n} & 0 & \dots & 0 & b_{00} & \dots & b_{0n} \\ a_{10} & \dots & a_{1n} & 0 & \dots & 0 & b_{10} & \dots & b_{1n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n0} & \dots & a_{nn} & 0 & \dots & 0 & b_{n0} & \dots & b_{nn} \end{array}\right)$$

The zeroes in the centre block account for the fact that there are no u_i^2 components in the first equation.

We can now formulate the rest of the matrix using this principle:

$$A = \left(\begin{array}{ccc|ccc|ccc} a_{00} & \dots & a_{0n} & 0 & \dots & 0 & b_{00} & \dots & b_{0n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n0} & \dots & a_{nn} & 0 & \dots & 0 & b_{n0} & \dots & b_{nn} \\ \hline 0 & \dots & 0 & a_{00} & \dots & a_{0n} & c_{00} & \dots & c_{0n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & a_{n0} & \dots & a_{nn} & c_{n0} & \dots & c_{nn} \\ \hline b_{00} & \dots & b_{n0} & c_{00} & \dots & c_{n0} & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ b_{0n} & \dots & b_{nn} & c_{0n} & \dots & c_{nn} & 0 & \dots & 0 \end{array}\right)$$

To put this into the matrix equation, we also define the vectors $\vec{v}$ and F:

$$\vec{v} = \begin{pmatrix} u_1^1 \\ u_2^1 \\ \vdots \\ u_n^1 \\ \hline u_1^2 \\ u_2^2 \\ \vdots \\ u_n^2 \\ \hline p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} \qquad F = \begin{pmatrix} d_0^1 \\ d_1^1 \\ \vdots \\ d_n^1 \\ \hline d_0^2 \\ d_1^2 \\ \vdots \\ d_n^2 \\ \hline 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

We can now numerically solve for $\vec{v}$ which gives us the velocity vector field $\vec{u} = (u_1, u_2)$ and the pressure function p.

6.5 Implementation and Results

We approach this problem by actually choosing $\vec{u}$ and p at the beginning. This allows us to numerically compute an appropriate $\vec{f}$ for this problem and gives us a correct solution to compare our numerical results to. To obtain $\vec{f}$, the specific functions we chose are

$$u_1 = 2\pi(1 - \cos(2\pi x))\sin(2\pi y)$$

$$u_2 = -2\pi(1 - \cos(2\pi y))\sin(2\pi x)$$

$$p = \sin(2\pi x)\sin(2\pi y).$$

In choosing these, we had to ensure that these equations also satisfy the divergence condition, which they do.

Another major part of implementing the code was constructing the matrix A. Notice how there are really only three non-zero different matrix blocks within this matrix (with some repeated, and some transposed). These blocks were calculated separately and then composed to obtain A. Figure 19 shows the matrix we obtain numerically. Note how in the non-zero blocks we can see a similar diagonal structure to what we have seen before. Observe also the plot of the pressure function $p(x, y)$ can be seen in Figure 20.

Now for the most interesting result, we obtain plots for the vector field $\vec{u}$. Figure 21 shows a vector field plot, with vectors scaled for visual clarity. In figure 22 we use a plotting software to obtain a "stream" plot of the vector field. This plot shows the direction of the flow of the fluid described by the vector field. We also plotted the individual results for the components of the vector which can be seen in figures 23 and 24. The figure on the left plots the horizontal velocity u_1, and the figure on the right plots the vertical velocity u_2. Once again, because we have chosen the solutions which we aim to obtain in advance, we can easily compare our numerical solutions to our expectations.

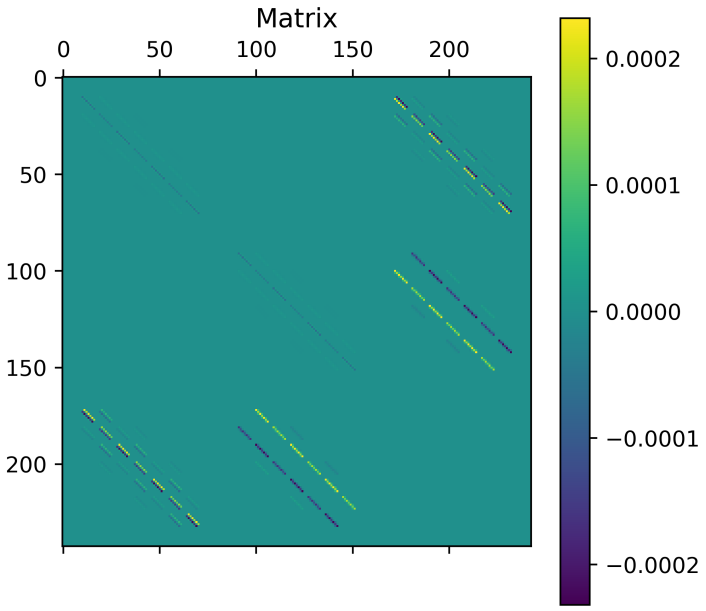


Figure 19: Matrix A for 81 nodes

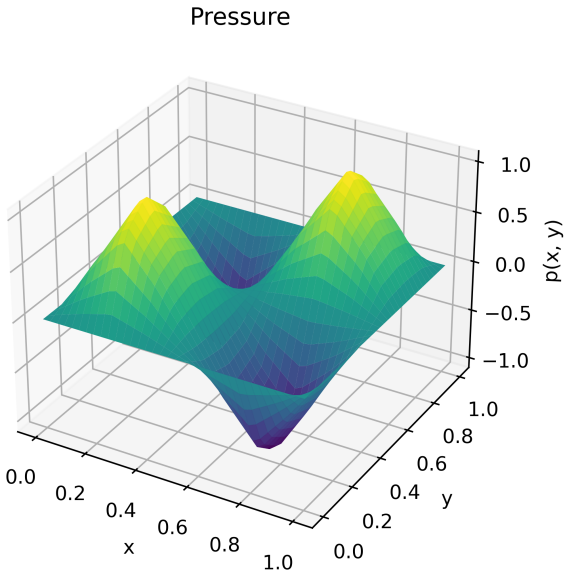


Figure 20: Numerically derived pressure function p

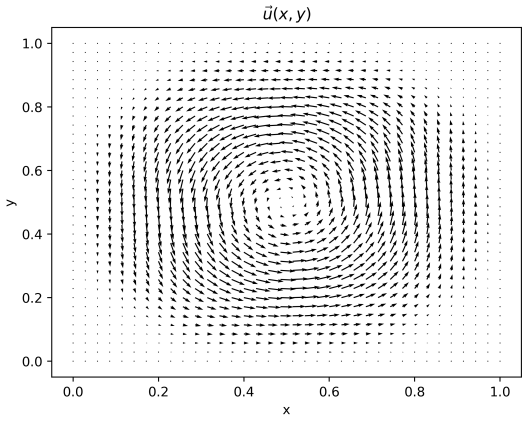


Figure 21: Vector plot of $\vec{u}$

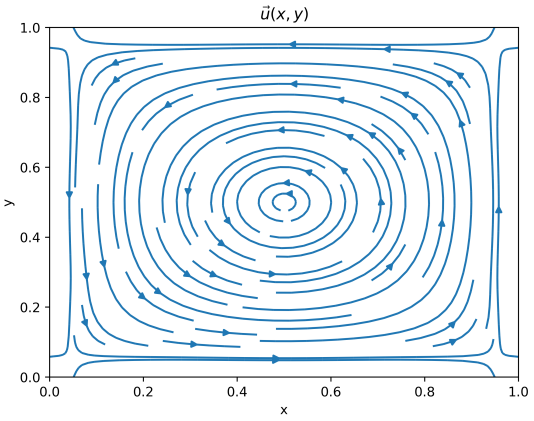


Figure 22: Stream plot of $\vec{u}$

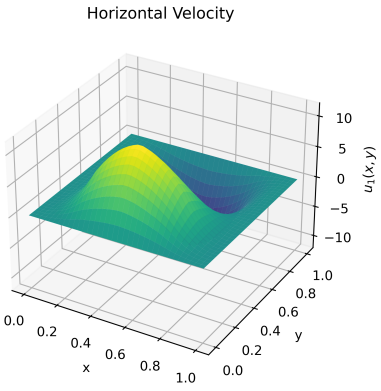


Figure 23: Horizontal velocity u_1

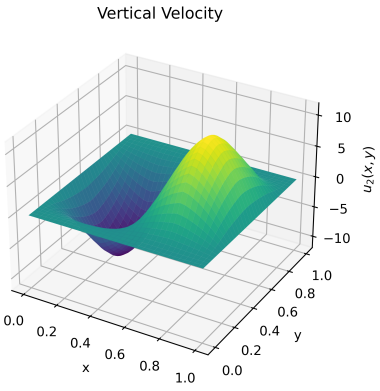


Figure 24: Vertical velocity u_2

6.6 Error Analysis

We once again use the L^2 norm to measure the error of our results for the Stokes problem. With an integral stepsize of 600 per element, we calculate the following errors.
On a 7×7 grid, we obtain an error of:

- ≈ 0.864872 for u_1
- ≈ 0.886772 for u_2
- ≈ 0.087308 for p

On a 9×9 grid, the errors we obtain for our solutions were:

- ≈ 0.717391 for u_1
- ≈ 0.717148 for u_2
- ≈ 0.060192 for p

Once again, it is clear to see that a more refined mesh leads to more accurate solutions.

7 Conclusion

In this project, we investigated the use of the Finite Element Method in solving a range of differential equations. The objective was to solve numerous problems with varying degrees of difficulty, to gain a deep insight into the structure of the Finite Element Method. The inspiration behind the chosen problems were the individual terms in the renowned Navier-Stokes equations. Initially we had planned to solve preliminary problems which would aid us in solving the Navier-Stokes equations in their entirety, but we soon came to realise that if we want to thoroughly understand the applications of the Finite Element Method to differential equations then we must exhaustively investigate each of these preliminary problems.

We began with a very simple ordinary differential equation, namely the one dimensional Poisson equation, to study the elemental concepts of the Finite Element Method. We then extended our understanding to the two dimensional Poisson problem, giving us the tools and experience we needed to solve the remaining multidimensional problems. We also realised that periodic boundary conditions are very important in the context of lots of differential equations so we started working on those but did not get to finalise the code. In principle, we learned that periodic boundary conditions involve making our basis functions periodic.

Another important class of differential equations in physics are time dependant equations, and thus we chose the classic problem of the heat equation. Both time dependent terms and Laplacian terms (as in the Poisson equations) appear in the Navier-Stokes equations, and we hoped that solving the aforementioned problems would be a stepping stone towards our initial goal. The time dependent term in the Navier-Stokes also contained a non-linear term, so we tried to solve a non-linear Poisson problem, but couldn't quite get a satisfactory implementation. Also, we realised that the solving of the Navier-Stokes equation by its very nature required more advanced techniques which apply specifically to the these equations. Nonetheless, there was a lot of interesting theory behind the investigation of this problem involving Gateaux derivatives, associated Taylor Series expansions and a clever Newton iteration method to converge to solutions.

A better approach was to solve a subcase of the Navier-Stokes problem, specifically the Stokes equations for slow fluid flow. This problem not only gave us a better understanding of how to solve vector equations using Finite Element Method, but also we obtained steady-state results for fluid flow as we had originally intended. We also learned more about convergence conditions, which led us to defining quadratic basis functions to solve the problem.

The Finite Element Method is quite mathematically involved, which was a point of major interest to us. We found that there is a general structure to using this method i.e. writing the problem in its weak formulation, discretising the domain and the equations using appropriate basis functions, and formulating the discretised weak form as a system of linear equations. However in each problem, we found that the beauty lies in the neat tricks which we use to simplify the process. For example, in the Stokes problem we had to come up with a creative way of writing three coupled equations into one matrix equation. We actually found that the resources online were quite limited in terms of explanation for this problem, so it took some effort to come up with the matrix formulation. This is partially the reason why we described the steps to create the matrix A in elaborate detail. Hopefully the next person to solve the Stokes equations using Finite Element Method can gain some insight from our formulations.

Overall, a lot was learned about Finite Element Analysis and its applications to differential equations. This project was a pleasure to investigate both from a theoretical and practical perspective. We hope to continue our exploration of the method in the future, and maybe one day we will have enough mastery and expertise to aim for that one million dollar prize.

8 Further Reading

In this report we included the most refined results which we obtained, however over the past few months, we went down many rabbit holes which were not included in this report. Some extra material which may be of interest is included in this section.

- A link to Patryk Drozd's Github page on everything associated with our investigation in this project, including a lot of material which is not explicitly written into this report.

  https://github.com/drozd324/Fluid-Sim

- From the Github, we wish to explicitly include the file solving a one dimensional Poisson problem with Neumann Boundary conditions.

  https://github.com/drozd324/Fluid-Sim/blob/main/code

- We also include a file containing a solution to the heat equation in which we construct our forcing function by choosing our solution in advance and working backwards. This was done to ensure that the code we have for solving the heat equation works.

  https://github.com/drozd324/Fluid-Sim/blob/main/code

- A short article about the mathematics of computing the Navier-Stokes equations with the Finite Element Method. This is one of the projects which we would love to explore in the future.

  https://jsdokken.com/dolfinx-tutorial/chapter2/navierstokes.html

- Notes on the implementation of periodic boundary conditions with the Finite Element Method.

  https://www.staff.uni-oldenburg.de/hannes.uecker/pde2path/tuts/pbctut.pdf

- As mentioned earlier, there was an attempt in solving a non- linear Poisson problem. If the reader chooses to check our implementation in the aforementioned Git Hub page, this resource contains notes on the mathematics behind solving a particular non linear Poisson problem.

  https://finite-element.github.io/8nonlinearproblems.html

9 Acknowledgements

We wish to give our thanks to the Theoretical Physics department for their support and guidance in this project, but also in our understanding of physics as a whole. We greatly appreciate the mathematical understanding Dr. Paul Watts provided us with which was needed in many aspects of our derivations. We are also grateful to Dr. John Brennan for providing clarity and insight into the computational aspects of this project which we struggled with. Most importantly, we thank Dr. Jonivar Skullerud for the opportunity to do this project. In particular, we acknowledge his support in allowing us to choose our own idea for the project, which proved to be an invaluable learning experience for us.

10 References

Note that many of these references have resources which are used throughout the entire project. In the report, each resource was referenced when it was firstly used.

[1] https://doi.org/10.1590/1806-9126-RBEF-2017-0239

[2] https://www.claymath.org/millennium/navier-stokes-equation/

[3] https://doi.org/10.1016/B978-1-4557-3141-1.50031-9

[4] https://doi.org/10.1016/B978-0-12-824117-2.00011-9

[5] https://doi.org/10.1016/B978-0-12-815601-8.50006-9

[6] https://ntrs.nasa.gov/citations/19840010609

[7] https://www.youtube.com/watch?v=P4lBRuY7pC4

[8] https://jsdokken.com/dolfinx-tutorial/chapter1/fundamentals.html

[9] Johnson, C. (n.d.). Numerical solution of partial differential equations by the finite element method. Courier Corporation.

[10] https://mathworld.wolfram.com/L2-Space.html

[11] https://fenicsproject.org/pub/tutorial/sphinx1/.ftut1003.html

[12] https://doi.org/10.1016/B978-0-7506-6722-7.X5030-9

[13] https://jsdokken.com/dolfinx-tutorial/chapter2/heatequation.html

[14] https://finite-element.github.io/

[15] https://doi.org/10.1007/978-3-030-23370-92

[16] https://doi.org/10.1016/B978-0-12-815489-2.00005-8

[17] https://people.sc.fsu.edu/ jburkardt/classes/fem2011/femstokes.pdf

11 Code

```
"""
This script solves the 1 dimensional poisson problem with dirichlet boundary
    conditions.

with solution

------ in LateX code ------

$u(x) = \frac{1}{2} x(x-1)

and forcing function

$f(x) = 1

"""

import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)
```

```
24
25 from tools import basis_functions as bf
26 from tools import norms as nrm
27
28 N = 200
29 steps = 10000 # number of steps in calculation of integrals
30
31 def solve_poisson(H):
32     """Container function for main part of code. The purpose of this is to create an
       easy way to
33     run solver with various parameters.
34
35     Args:
36         H (int): number of nodes
37
38     Returns:
39         tuple: solution to equation, solution matrix A, solution matrix F, error
40     """
41
42     vertices = list(np.linspace(0, 1, H))
43     elements = [[vertices[i], vertices[i+1]] for i in range(H-1)]
44     h = vertices[1] - vertices[0]
45
46     # defining basis functions and their derivatives. This was done initially to swap
       out linear basis functions to quadratic basis function just because we can
47     def basis_func(vert, x):
48         return bf.phi(vert, x, h)
49
50     def d_basis_func(vert, x):
51         return bf.grad_phi(vert, x, h)
52
53
54     def force(x):
55         return -1
56
57     # defining matrix to calculate entries for
58     A = np.zeros((H, H))
59     F = np.zeros((H))
60
61     # functions for calculating entries of matrices
62     def a(vert0, vert1, x):
63         return d_basis_func(vert0, x) * d_basis_func(vert1, x)
64
65     def f(vert0, x):
66         return basis_func(vert0, x) * force(x)
67
68     # main bit of code iterating over elements of mesh (here a 1dim line) and
       calcualting entries of the matrices
69     for ele in elements:
70         x = np.linspace(ele[0], ele[1], steps)
71
72         for vert0 in ele:
73             j = vertices.index(vert0)
74             F[j] = F[j] + np.trapz(f(vert0, x), x)
75
76             for vert1 in ele:
77                 i = vertices.index(vert1)
78                 A[i, j] = A[i, j] + np.trapz(a(vert0, vert1, x), x)
79
80     # final calculation of solution
81     A_inverted = np.linalg.pinv(A)
82     solution = np.matmul(A_inverted, F)
83
84     vertices = np.array(vertices)
85     error = nrm.l_squ_norm((((vertices**2)/2) - (vertices/2)) - solution, vertices)
```

```
86
87      x = np.linspace(0, 1, N)
88      u = bf.conv_sol(solution, x, bf.hat, vertices, h)
89
90
91      return u, A, F, error
92
93  poisson_sols = []
94  vert_num = [3, 5, 20] # number of vertices to solve the equation over
95  for num in vert_num:
96      sol = solve_poisson(num)
97      poisson_sols.append(sol)
98
99  x = np.linspace(0, 1, N)
100 for i, u in enumerate(poisson_sols):
101     plt.plot(x, u[0], label=f"{vert_num[i]} vertices")
102 plt.xlabel("x")
103 plt.ylabel("u(x)")
104 plt.plot(x, ((x**2)/2) - (x/2), label="Analytical Solution")
105 plt.legend()
106 plt.title(r'Solution for $-u^{\prime\prime}(x) = 1$')
107 #plt.savefig(f"(Poisson_1d)_(lin_sol)_(x)", dpi=500)
108 plt.show()
109
110 for i in range(len(vert_num)):
111     plt.matshow(poisson_sols[i][1])
112     plt.title(r'Solution Matrix for $-u^{\prime\prime}(x) = 1$' + f"   ,vert num = {
        vert_num[i]}")
113     plt.colorbar()
114     plt.show()
115     #plt.savefig(f"(Poisson_1d)_(lin_matrix_A)_(vertex_num_{H})")
116
117 """ a method of measuring how accurate our solutions are would be to use a norm on
        functions.
118 There is an $L^2$ norm which is used to comapare functions. We would simply take the
        difference
119 of our solution and its analytical solution and throw it into this norm.
120 """
121 print(f"L squared norm error for 20 vertices = {poisson_sols[2][3]}")
```

Listing 1: poisson_1dim_ele_dirichlet.py

```
1   """
2   Script solving the 2d poisson problem with dirichlet boudary conditions.
3
4   With manufactured forcing function
5
6   --------- in LateX code-----------
7
8   $f(x, y) = -2 \pi^2 sin(\pi x)sin(\pi y)$
9
10  and solution
11
12  $u(x, y) = -sin(\pi x)sin(\pi y)
13
14  """
15
16  import numpy as np
17  import matplotlib.pyplot as plt
18  import time
19
20  import os
21  import sys
22  current_dir = os.getcwd()
23  top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
```

```
sys.path.append(top_dir)

from tools import basis_functions as bf
from tools import vector_products as vp
from tools import norms as nrm

# main parameters to change
steps = 1000 # accuracy or steps in integrals, make this smaller for significantly quicker run time
H = 5 # $H^2$ is the number of nodes in the mesh, tweak this parameter to got a denser mesh
# important to increase steps along side H

# defining mesh
x = np.linspace(0, 1, H)
y = np.linspace(0, 1, H)
h = x[1] - x[0]
vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()
elements = [[[x[i]  , y[j]  ],
             [x[i+1], y[j]  ],
             [x[i]  , y[j+1]],
             [x[i+1], y[j+1]]] for i in range(H-1) for j in range(H-1)]


# defining functions for integrands
def force(x):
    return -2*(np.pi**2)*np.sin(np.pi*x[0])*np.sin(np.pi*x[1])

def f(vert, x):
    return bf.phi_2d(vert, x, h) * force(x)

def a(vert0, vert1, x):
    return vp.grad_dot_grad_phi2d(vert0, vert1, x, h)

A = np.zeros((H**2, H**2))
F = np.zeros((H**2))

t0 = time.time()
for k, ele in enumerate(elements):
    x0 = np.linspace(ele[0][0], ele[3][0], steps)
    y0 = np.linspace(ele[0][1], ele[3][1], steps)
    X0, Y0 = np.meshgrid(x0, y0)

    percentage = round(100 * ((k)/(len(elements)-1)), 1)
    print(f"{percentage}%", end="\r")

    for vert0 in ele:
        j = vertices.index(vert0)
        F[j] = F[j] + np.trapz(np.trapz( f(vert0, (X0, Y0)) , y0, axis=0), x0, axis=0)

        for vert1 in ele:
            i = vertices.index(vert1)
            A[j, i] = A[j, i] + np.trapz(np.trapz( a(vert0, vert1, (X0, Y0)) , y0, axis=0), x0, axis=0)

t1 = time.time()
print(f"time taken: {(t1-t0)/60} minutes")

# this is solving the matrix equation Au = F
solution = np.matmul(np.linalg.pinv(A), F)

# the rest is plotting
N = 3*H
x0 = np.linspace(0, 1, N)
y0 = np.linspace(0, 1, N)
```

```
X0, Y0 = np.meshgrid(x0, y0)

fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_surface(X0, Y0, bf.conv_sol(solution, (X0, Y0), bf.phi_2d, vertices, h), cmap=
    "viridis")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u(x,y)')
ax.set_title(r"$ \nabla^2 u(x,y) = -2 \pi^2sin(\pi x)sin(\pi y) $")
#plt.savefig(f"(Poisson_2d)_(vertex_num_{H**2})", dpi=500)

plt.matshow(A)
plt.colorbar()
#plt.savefig(f"(Poisson_2d)_(mat_A)_(vertex_num_{H**2})", dpi=500)
plt.show()

#error check
math_sol = lambda x: -np.sin(np.pi*x[0])*np.sin(np.pi*x[1])
ele_sol = bf.conv_sol(solution, (X0, Y0), bf.phi_2d, vertices, h)
print(f"L squared norm error with {H**2} nodes = {nrm.l_squ_norm_2d(ele_sol - math_sol
    ((X0, Y0)), (x0, y0))}")
```

Listing 2: poisson_2dim_ele_dirichlet.py

```
"""
Script solving the 2d poisson problem with neumann boundary conditions.

With manufactured forcing function

--------- in LateX code-----------

$f(x, y) = -2y

neumann boundary conditions

$\del u \cdotp \hat{n} = -x(x-1)$

and solution

$u(x, y) = -yx(x - 1)

"""

import numpy as np
import matplotlib.pyplot as plt
import time

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf
from tools import vector_products as vp
from tools import norms as nrm

# main parameters to change
steps = 200 # accuracy or steps in integrals, make this smaller for significantly
    quicker run time
H = 7 # $H^2$ is the number of nodes in the mesh, tweak this parameter to got a denser
     mesh

# defining mesh
```

```
x = np.linspace(0, 1, H)
y = np.linspace(0, 1, H)
h = x[1] - x[0]
vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()
elements = [[[x[i]  , y[j]  ],
             [x[i+1], y[j]  ],
             [x[i]  , y[j+1]],
             [x[i+1], y[j+1]]] for i in range(H-1) for j in range(H-1)]

# defining functions for integrands
def force(x):
    x, y = x
    return -2*y

def n(x):
    x, y = x
    return  -x*(x-1)

def f(vert, x):
    return bf.phi_and_hat_2d(vert, x, h, [0]) * force(x)

def a(vert0, vert1, x):
    return vp.gdg_phi_and_hat_2d(vert0, vert1, x, h, [0])

def neumann(vert, x):
    x, y = x
    int1 = np.trapz( n((x, 0)) * bf.phi_and_hat_2d(vert, (x, 0), h, []) , x, axis=0)
    int2 = np.trapz( n((x, 1)) * bf.phi_and_hat_2d(vert, (x, 1), h, []) , x, axis=0)
    return int1 + int2

A = np.zeros((H**2, H**2))
F = np.zeros((H**2))


t0 = time.time()
for k, ele in enumerate(elements):
    x0 = np.linspace(ele[0][0], ele[3][0], steps)
    y0 = np.linspace(ele[0][1], ele[3][1], steps)
    X0, Y0 = np.meshgrid(x0, y0)

    percentage = round(100 * ((k)/(len(elements)-1)), 1)
    print(f"{percentage}%", end="\r")

    for vert0 in ele:
        j = vertices.index(vert0)
        F[j] = F[j] - np.trapz(np.trapz( f(vert0, (X0, Y0)) , y0, axis=0), x0, axis=0)
     + neumann(vert0, (x0, y0))

        for vert1 in ele:
            i = vertices.index(vert1)
            A[j, i] = A[j, i] + np.trapz(np.trapz( a(vert0, vert1, (X0, Y0)) , y0,
    axis=0), x0, axis=0)


t1 = time.time()
print(f"time taken: {(t1-t0)/60} minutes")

# this is solving the matrix equation Au = F
solution = np.matmul(np.linalg.pinv(A), F)

# the rest is plotting
N = 3*H
x0 = np.linspace(0, 1, N)
y0 = np.linspace(0, 1, N)
X0, Y0 = np.meshgrid(x0, y0)
```

```

fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_surface(X0, Y0, bf.conv_sol(solution, (X0, Y0), bf.hat_2d, vertices, h), cmap="viridis")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u(x,y)')
#ax.set_title(r"$ \nabla^2u(x,y) = -2y $ with neumann bdry $n(x,y) = -x(x-1)$")
#plt.savefig(f"(Poisson_2d)_(neumann)_(vertex_num_{H**2})", dpi=500)

plt.matshow(A)
plt.colorbar()
plt.savefig(f"(Poisson_2d)_(mat)_(neumann)_(vertex_num_{H**2})", dpi=500)
plt.show()

#error check
math_sol = lambda x: -np.sin(np.pi*x[0])*np.sin(np.pi*x[1])
ele_sol = bf.conv_sol(solution, (X0, Y0), bf.phi_2d, vertices, h)
print(f"L squared norm error with {H**2} nodes = {nrm.l_squ_norm_2d(ele_sol - math_sol((X0, Y0)), (x0, y0))}")
```

Listing 3: poisson_2dim_ele_neumann.py

```

"""
This a file is for solving the heat equation with

------ in LateX code --------

$u(x, y, 0) = sin(\pi x) sin(\pi y)$

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

-----------------------------

"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf
from tools import vector_products as vp

steps = 100 # steps in integrals

# defining mesh
H = 7
x = np.linspace(0, 1, H)
y = np.linspace(0, 1, H)
X, Y = np.meshgrid(x, y)
h = x[1] - x[0]
vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()
elements = [[[x[i]  , y[j]  ],
             [x[i+1], y[j]  ],
             [x[i]  , y[j+1]],
             [x[i+1], y[j+1]]] for i in range(H-1) for j in range(H-1)]

```

```
# declaring time stepping necessities
dt = .01 # change in time
time_steps = 50 # amount of iterations we want
u_0 = [np.sin(np.pi*x[0]) * np.sin(np.pi*x[1]) for x in vertices] # initial condtions at time=0
u_sols = [u_0] # list for keeping track of evolution of system
A_sols = []

def force(x, t):
    return 0

# functions for calculating the entries of matrices
def f(vert, x, u, time):
    return (u + (dt * force(x, time))) * bf.phi_2d(vert, x, h)

def a(vert0, vert1, x):
    return (bf.phi_2d(vert0, x, h)*bf.phi_2d(vert1, x, h)) + (dt*(vp.grad_dot_grad_phi2d(vert0, vert1, x, h)))

t0 = time.time()
for t in range(time_steps): #time stepping loop
    A = np.zeros((H**2, H**2))
    F = np.zeros((H**2))

    # main code, iterating over elements and calculating entries of matrix
    for k, ele in enumerate(elements):
        # creating domain of element to iterate over
        x0 = np.linspace(ele[0][0], ele[3][0], steps)
        y0 = np.linspace(ele[0][1], ele[3][1], steps)
        X0, Y0 = np.meshgrid(x0, y0)

        u = bf.conv_sol(u_sols[t], (X0, Y0), bf.phi_2d, vertices, h)

        percentage = round(100 * ((k)/(len(elements)-1)), 1)
        print(f"time {t} and {percentage}%", end="\r")

        for vert0 in ele:
            j = vertices.index(vert0)
            F[j] = F[j] + np.trapz(np.trapz( f(vert0, (X0, Y0), u, t*dt) , y0, axis=0), x0, axis=0)

            for vert1 in ele:
                i = vertices.index(vert1)
                A[j, i] = A[j, i] + np.trapz(np.trapz( a(vert0, vert1, (X0, Y0)) , y0, axis=0), x0, axis=0)


    solution_t = np.matmul(np.linalg.pinv(A), F)
    u_sols.append(solution_t)
    A_sols.append(A)


t1 = time.time()
print(f"time taken: {(t1-t0)/60} minutes")

# creating mesh with higher resolution for plotting
N = 4*H
x = np.linspace(0, 1, N)
y = np.linspace(0, 1, N)
X, Y = np.meshgrid(x, y)

func_u_sols = []
for i in range(len(u_sols)):
    func_u_sols.append(bf.conv_sol(u_sols[i], (X, Y), bf.phi_2d, vertices, h))

```

```
"""--------------- PLOTTING --------------------"""

# plot and animation
len_u_sols = len(func_u_sols)
frn = len_u_sols
fps = frn//3

def change_plot(frame_number, func_u_sols, plot):
   plot[0].remove()
   plot[0] = ax.plot_surface(X, Y, func_u_sols[frame_number], cmap="plasma", vmin=0,
    vmax=1)

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
plot = [ax.plot_surface(X, Y, func_u_sols[0])]
ax.set_zlim(0, 1)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("u(x, y, t)")
ax.set_title(r"$ \frac{\partial u}{\partial t} = \nabla^2 u $ with $u(x,y,0) = sin(\pi
    x)sin(\pi y)$")

ani = animation.FuncAnimation(fig, change_plot, frn, fargs=(func_u_sols, plot),
    interval=1000 / fps)
ani.save('animation.gif',writer='PillowWriter',fps=fps, dpi=400)
plt.show()

# plot of matrix for first iteration
plt.matshow(A_sols[0])
plt.title("Matrix for first iteration")
plt.colorbar()
plt.show()

# creating a sequence of images for report
plt.clf()
iter1 = 0
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.axes.set_zlim3d(bottom=0, top=1)
ax.plot_surface(X, Y, func_u_sols[iter1], cmap="plasma", vmin=0, vmax=1)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel(f'$u(x, y, {iter1*dt})$')
#plt.savefig(f"heat_equ_iter{iter1}", dpi=400)

plt.clf()
iter2 = int(len_u_sols/12)
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.axes.set_zlim3d(bottom=0, top=1)
ax.plot_surface(X, Y, func_u_sols[iter2], cmap="plasma", vmin=0, vmax=1)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel(f'$u(x, y, {iter2*dt})$')
#plt.savefig(f"heat_equ_iter{iter2}", dpi=400)

plt.clf()
iter3 = int(len_u_sols/6)
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.axes.set_zlim3d(bottom=0, top=1)
ax.plot_surface(X, Y, func_u_sols[iter3], cmap="plasma", vmin=0, vmax=1)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel(f'$u(x, y, {iter3*dt})$')
```

```
#plt.savefig(f"heat_equ_iter{iter3}", dpi=400)

plt.clf()
iter4 = time_steps
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.axes.set_zlim3d(bottom=0, top=1)
ax.plot_surface(X, Y, func_u_sols[iter4], cmap="plasma", vmin=0, vmax=1)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel(f'$u(x, y, {iter4*dt})$')
#plt.savefig(f"heat_equ_iter{iter4}", dpi=400)
```

Listing 4: heat_equation_2+1dim_ele.py

```
"""
Script for solving the stokes equation with a manufactured forcing functions for the
    solutions.

------ in LateX code ------

$u_1 =  2\pi(1 - cos(2\pi x)sin(2\pi y)$
$u_2 = -2\pi(1 - cos(2\pi y)sin(2\pi x)$
$p   = sin(2 \pi x)sin(2 \pi y)$

---------------------------

"""

import numpy as np
import matplotlib.pyplot as plt
import time

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf
from tools import vector_products as vp
from tools import norms as nrm

# main parameters to tweak
H = 3 + (2*3) # we need a specific amount of nodes to make use of quadratic elements,
    here in the form of; 3 + 2*n for n a natural number
steps = 600 # steps in integrals over each element
# as you increase H make sure to increase steps

# defining mesh
x = np.linspace(0, 1, H)
y = np.linspace(0, 1, H)
X, Y = np.meshgrid(x, y)
h = x[1] - x[0]
vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()

# defining quadratic elements
quad_elements = []
for i in list(np.arange(0, H-1, 2)):
    for j in list(np.arange(0, H-1, 2)):

        element = []
        for a in range(0, 3):
            for b in range(0, 3):
                element.append([x[i+a], y[j+b]])
```

```
49
50          quad_elements.append(element)
51
52  # manufacturing solutions to stokes equation
53  mu = 1 # viscocity
54  u_1 = lambda x: 2*np.pi*(1 - np.cos(2*np.pi*x[0]))*np.sin(2*np.pi*x[1])
55  u_2 = lambda x: -2*np.pi*(1 - np.cos(2*np.pi*x[1]))*np.sin(2*np.pi*x[0])
56  p = lambda x: np.sin(2*np.pi*x[0]) * np.sin(2*np.pi*x[1])
57
58  # first derivatives
59  dx_p = lambda x: np.gradient(p(x), h, edge_order=2)[1]
60  dy_p = lambda x: np.gradient(p(x), h, edge_order=2)[0]
61
62  dx_u1 = lambda x: np.gradient(u_1(x), h, edge_order=2)[1]
63  dy_u1 = lambda x: np.gradient(u_1(x), h, edge_order=2)[0]
64
65  dx_u2 = lambda x: np.gradient(u_2(x), h, edge_order=2)[1]
66  dy_u2 = lambda x: np.gradient(u_2(x), h, edge_order=2)[0]
67
68  # second derivatives
69  ddx_u1 = lambda x: np.gradient(dx_u1(x), h, edge_order=2)[1]
70  ddy_u1 = lambda x: np.gradient(dy_u1(x), h, edge_order=2)[0]
71
72  ddx_u2 = lambda x: np.gradient(dx_u2(x), h, edge_order=2)[1]
73  ddy_u2 = lambda x: np.gradient(dy_u2(x), h, edge_order=2)[0]
74
75  # declaring appropriate forcing functions. Change these to anything you want if you
        want to test the code
76  f_1 = lambda x: (mu*(ddx_u1(x) + ddy_u1(x))) - dx_p(x)
77  f_2 = lambda x: (mu*(ddx_u2(x) + ddy_u2(x))) - dy_p(x)
78
79  # declaring appropriate funtions for block matrices
80  gdg         = lambda vert0, vert1, x, h: - mu * vp.grad_dot_grad(bf.psi_2d(vert0, x, h)
        , bf.psi_2d(vert1, x, h) , h)
81  hat_dx_psi = lambda vert0, vert1, x, h: -bf.phi_2d(vert0, x, h) * (np.gradient(bf.
        psi_2d(vert1, x, h), h)[1])
82  hat_dy_psi = lambda vert0, vert1, x, h: -bf.phi_2d(vert0, x, h) * (np.gradient(bf.
        psi_2d(vert1, x, h), h)[0])
83
84  t0 = time.time()
85
86  # block matrix calculations
87  mat_blocks = []
88  mat_funcs  = [gdg, hat_dx_psi, hat_dy_psi]
89  for ka, a in enumerate(mat_funcs):
90      BLOCK = np.zeros((H**2, H**2))
91
92      for k, ele in enumerate(quad_elements):
93          # creating domain for integration
94          x0 = np.linspace(ele[0][0], ele[-1][0], steps)
95          y0 = np.linspace(ele[0][1], ele[-1][1], steps)
96          X0, Y0 = np.meshgrid(x0, y0)
97
98          # keep track of code running
99          percentage = round(100 * ((k)/(len(quad_elements)-1)), 1)
100         print(f"mat_{ka} {percentage}%", end="\r")
101
102         # main iteration loop
103         for vert0 in ele:
104             j = vertices.index(vert0)
105             for vert1 in ele:
106                 i = vertices.index(vert1)
107                 BLOCK[j, i] = BLOCK[j, i] + np.trapz(np.trapz( a(vert0, vert1, (X0, Y0
        ), h) , y0, axis=0), x0, axis=0)
108
```

```
    mat_blocks.append(BLOCK)

# block vector calculations
vect_blocks = []
force_funcs = [f_1, f_2]
for kf, f in enumerate(force_funcs):
    BLOCK = np.zeros((H**2))

    for k, ele in enumerate(quad_elements):
        # creating domain for integration
        x0 = np.linspace(ele[0][0], ele[-1][0], steps)
        y0 = np.linspace(ele[0][1], ele[-1][1], steps)
        X0, Y0 = np.meshgrid(x0, y0)

        # keep track of code running
        percentage = round(100 * ((k)/(len(quad_elements)-1)), 1)
        print(f"force_{kf} {percentage}%", end="\r")

        # main iteration loop
        for vert0 in ele:
            j = vertices.index(vert0)
            BLOCK[j] = BLOCK[j] + np.trapz(np.trapz( (bf.hat_2d(vert0, (X0, Y0), h)) *
     f((X0, Y0)) , y0, axis=0), x0, axis=0)

    vect_blocks.append(BLOCK)

t1 = time.time()
print(f"time taken: {round((t1-t0)/60, 2)} minutes
                        ")

# assembly of blocks
zero_mat = np.zeros((H**2, H**2))
zero_vect = np.zeros((H**2))

reshaped_mat_blocks = [[mat_blocks[0]   , zero_mat       , mat_blocks[1]],
                       [zero_mat        , mat_blocks[0]  , mat_blocks[2]],
                       [mat_blocks[1].T, mat_blocks[2].T, zero_mat     ]]

reshaped_vect_blocks = [vect_blocks[0], vect_blocks[1], zero_vect]

A = np.block(reshaped_mat_blocks)
F = np.block(reshaped_vect_blocks)

# calculation of solution
solution = np.matmul(np.linalg.pinv(A), F)

# creating mesh with higher resolution for plotting
N = 4*H
x = np.linspace(0, 1, N)
y = np.linspace(0, 1, N)
X, Y = np.meshgrid(x, y)

# dissassembling solution vector
u1_sol = bf.conv_sol(solution[       0 : H**2    ], (X, Y), bf.psi_2d, vertices, h)
u2_sol = bf.conv_sol(solution[H**2     : 2*(H**2)], (X, Y), bf.psi_2d, vertices, h)
p_sol  = bf.conv_sol(solution[2*(H**2) :         ], (X, Y), bf.hat_2d, vertices, h)

# plotting
plt.matshow(A)
plt.colorbar()
plt.title("Matrix")
plt.savefig("Vector_pde_matrix", dpi=500)

fig, ax = plt.subplots()
ax.quiver(X, Y, u1_sol, u2_sol)
```

```
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title(r"$\vec{u}(x, y)$")
plt.savefig("vector fiel", dpi=500)

fig, ax = plt.subplots()
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title(r"$\vec{u}(x, y)$")
ax.streamplot(X, Y, u1_sol, u2_sol)
plt.savefig("stream", dpi=500)

fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel(r"$u_1(x, y)$")
ax.set_title("Horizontal Velocity")
ax.plot_surface(X, Y, u1_sol, cmap="viridis")
plt.savefig("horizontal velocity", dpi=500)

fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel(r"$u_2(x, y)$")
ax.set_title("Vertical Velocity")
ax.plot_surface(X, Y, u2_sol, cmap="viridis")
plt.savefig("vertical velocity", dpi=500)

fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("p(x, y)")
ax.set_title("Pressure")
ax.plot_surface(X, Y, p_sol, cmap="viridis")
plt.savefig("pressure", dpi=500)

plt.show()

#error check
error_u1_sol = nrm.l_squ_norm_2d(u_1((X, Y)) - u1_sol, (x, y))
error_u2_sol = nrm.l_squ_norm_2d(u_2((X, Y)) - u2_sol, (x, y))
error_p_sol  = nrm.l_squ_norm_2d(p((X, Y))   - p_sol , (x, y))

error_sum = error_u1_sol + error_u2_sol + error_p_sol
print(f"{H**2} nodes per function")
print(f"u1 error = {error_u1_sol}")
print(f"u2 error = {error_u2_sol}")
print(f"p error = {error_p_sol}")
```

Listing 5: stokes_2dim_ele.py

```
"""
Plotting of 2dim hat function
"""

import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
```

```
sys.path.append(top_dir)

from tools import basis_functions as bf

N = 31
x = np.linspace(0, 1, N)
y = np.linspace(0, 1, N)
X, Y = np.meshgrid(x, y)

fig = plt.figure()
ax = plt.axes(projection ='3d')

ax.plot_surface(X, Y, bf.hat_2d((0.5, 0.5), (X, Y), .25), cmap="viridis")
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel(r'$\phi_{\frac{1}{2},\frac{1}{2}}(x,y)$')
#plt.savefig("hat_basis_2d.png", dpi=500)
plt.show()
```

Listing 6: 2d_hat_plot.py

```
"""
Plotting of 2dim quadratic basis functions
"""

import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf

N = 200
x = np.linspace(0, 1, N)
y = np.linspace(0, 1, N)
X, Y = np.meshgrid(x, y)

h = 0.1

pair_1 = (0.5, 0.6)
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_surface(X, Y, bf.quad_2d(pair_1, (X, Y), h), cmap="viridis")
ax.set_xlabel('x')
ax.set_ylabel('y')
#plt.savefig(f"quad_basis_2d_{pair_1}.png", dpi=500)
plt.show()

pair_2 = (0.5, 0.5)
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_surface(X, Y, bf.quad_2d(pair_2, (X, Y), h), cmap="viridis")
ax.set_xlabel('x')
ax.set_ylabel('y')
#plt.savefig(f"quad_basis_2d_{pair_2}.png", dpi=500)
plt.show()

pair_3 = (0.6, 0.5)
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_surface(X, Y, bf.quad_2d(pair_3, (X, Y), h), cmap="viridis")
```

```
ax.set_xlabel('x')
ax.set_ylabel('y')
#plt.savefig(f"quad_basis_2d_{pair_3}.png", dpi=500)
plt.show()

pair_4 = (0.6, 0.6)
fig = plt.figure()
ax = plt.axes(projection ='3d')
ax.plot_surface(X, Y, bf.quad_2d(pair_4, (X, Y), h), cmap="viridis")
ax.set_xlabel('x')
ax.set_ylabel('y')
#plt.savefig(f"quad_basis_2d_{pair_4}.png", dpi=500)
plt.show()
```

Listing 7: 2d_quad_plot.py

```
"""
Plotting of sum of hat basis fuctions to show linear interpolation between their peaks
"""

import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf


N = 1000
x = np.linspace(0, 1, N)

H = 5
vertices = np.linspace(0, 1, H)
h = vertices[1] - vertices[0]

func1 = 1 * bf.phi(.25, x, h)
func2 = 3 * bf.phi(.5, x, h)
func3 = 2 * bf.phi(.75, x, h)
func_sum = func1 + func2 + func3

plt.plot(x, func_sum, label="sum of bases", linestyle="-")
plt.plot(x, func1, label=r"$1\phi_{0.25}(x)$", linestyle="--")
plt.plot(x, func2, label=r"$3\phi_{0.5}(x)$" , linestyle="-.")
plt.plot(x, func3, label=r"$2\phi_{0.75}(x)$", linestyle=":")
plt.xlabel("x")
plt.legend()
#plt.savefig("linear_basis_sum")
plt.show()
```

Listing 8: plot_hat_basis_sum.py

```
"""
Plotting of hat basis function and its derivative
"""

import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
```

```
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf


N = 200
x = np.linspace(0, 1, N)
h = .3

plt.plot(x, bf.hat(.5, x, h), label=r"$\phi_{0.5}(x)$")
plt.plot(x, bf.grad_hat(.5, x, h), label=r"$\phi_{0.5}^\prime(x)$", linestyle="--")
plt.legend()
plt.title("the hat function and its derivative")
plt.xlabel("x")
#plt.savefig("linear_basis")
plt.show()
```

Listing 9: plot_hat_basis.py

```
"""
Plotting of "even" quadratic basis function
"""


import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf

N = 1000
x = np.linspace(0, 1, N)

H = 6
vertices = np.linspace(0, 1, H)
h = vertices[1] - vertices[0]

func1 = 1 * bf.quad_e(.5, x, h)

plt.plot(x, func1)
plt.xlabel("x")
#plt.savefig("quad_basis_even")
plt.show()
```

Listing 10: plot_quad_basis_even.py

```
"""
Plotting of "odd" quadratic basis function
"""

import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

```

```
from tools import basis_functions as bf

N = 1000
x = np.linspace(0, 1, N)

H = 6
vertices = np.linspace(0, 1, H)
h = vertices[1] - vertices[0]

func2 = 1 * bf.quad_o(.5, x, h)

plt.plot(x, func2)
plt.xlabel("x")
#plt.savefig("quad_basis_odd")
plt.show()
```

Listing 11: plot_quad_basis_odd.py

```
"""
Plotting of a sum of quadratic basis functions scaled by some numbers to show
    quadratic interpolation for the element
"""

import numpy as np
import matplotlib.pyplot as plt

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf

N = 1000
x = np.linspace(0, 1, N)

H = 3
vertices = np.linspace(0, 1, H)
h = vertices[1] - vertices[0]

func1 = 1 * bf.quad(0, x, h)
func2 = 3 * bf.quad(.5, x, h)
func3 = 3 * bf.quad(1, x, h)
func_sum = func1 + func2 + func3

plt.plot(x, func1, label=r"$ 1 \psi_{0}(x)$", linestyle="--")
plt.plot(x, func2, label=r"$ 3 \psi_{0.5}(x)$", linestyle="--")
plt.plot(x, func3, label=r"$ 3 \psi_{1}(x)$", linestyle="--")
plt.plot(x, func_sum, label="sum of bases")

plt.xlabel("x")
plt.legend()
#plt.savefig("quad_basis")
plt.show()
```

Listing 12: plot_quad_basis_funky.py

```
"""
Plotting of a sum of quadratic basis functions on one element
"""

import numpy as np
import matplotlib.pyplot as plt

```

```
import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf

N = 1000
x = np.linspace(0, 1, N)

H = 3
vertices = np.linspace(0, 1, H)
h = vertices[1] - vertices[0]

func1 = 1 * bf.quad(0, x, h)
func2 = 1 * bf.quad(.5, x, h)
func3 = 1 * bf.quad(1, x, h)
func_sum = func1 + func2 + func3

plt.plot(x, func1, label=r"$ \psi_{0}(x)$", linestyle="--")
plt.plot(x, func2, label=r"$ \psi_{0.5}(x)$", linestyle="--")
plt.plot(x, func3, label=r"$ \psi_{1}(x)$", linestyle="--")
plt.plot(x, func_sum, label="sum of bases")

plt.xlabel("x")
plt.legend()
#plt.savefig("quad_basis")
plt.show()
```

Listing 13: plot_quad_basis_sum.py

```
import numpy as np

"""-------------------- LINEAR BASIS FUNCTIONS ----------------------"""

epsilon = 1e-3

def hat(i, x, h):
    """hat function with peak at i.

    Args:
        i (float): position of peak
        x (float): variable

    Returns:
        float:
    """
    return np.piecewise(x, [x <= (i-h), (x > (i-h))&(x <= i)        , (x > (i))&(x < (i+h))           , x>=(i+h)],
                           [0          , lambda x: (x/h) + (1-(i/h)) , lambda x: (-x/h) + (1+(i/h))    , 0                 ])


def grad_hat(i, x, h):
    """derivative of hat function with peak at i

    Args:
        i (float): position of peak
        x (float): variable

    Returns:
        float:
    """
    return np.piecewise(x, [x <= (i-h), (x > (i-h))&(x <= i), (x > (i))&(x < (i+h)), x
```

```
    >=(i+h)],
                                [0          , 1/h                   , -1/h
     , 0       ])


def hat_2d(i, x, h):
    return hat(i[0], x[0], h) * hat(i[1], x[1], h)

def phi_and_hat_2d(i, x, h, hat_bdry=[]):
    return phi(i[0], x[0], h) * phi(i[1], x[1], h, hat_bdry)


def phi(i, x, h, boundary=[0, 1]):
    """Piecewise linear basis function

    Args:
        i (float): position of peak
        x (float): variable
        boundary (tuple): boundary conditions on

    Returns:
        float:
    """

    if (i == np.array(boundary)).any(): #np.abs(i - 0) < epsilon or np.abs(i - 1) <
    epsilon:
        return 0*x
    else:
        return hat(i, x, h)

def grad_phi(i, x, h, boundary=[0, 1]):
    """derivative of hat function with peak at i

    Args:
        i (float): position of peak
        x (float): variable

    Returns:
        float:
    """
    if (i == np.array(boundary)).any(): #np.abs(i - 0) < epsilon or np.abs(i - 1) <
    epsilon:
        return 0*x
    else:
        return np.piecewise(x, [x <= (i-h), (x > (i-h))&(x <= i), (x > (i))&(x < (i+h)
    ), x>=(i+h)],
                                [0          , 1/h                   , -1/h
     , 0       ])

def phi_2d(i, x, h):
    """2 dimensional hat function with peak at i.

    Args:
        i (tuple): position of peak
        x (tuple): variable

    Returns:
        float:
    """

    return phi(i[0], x[0], h) * phi(i[1], x[1], h)

# some derivatives of previously declared functions
def dx_phi_2d(i, x, h):
    return grad_phi(i[0], x[0], h) * phi(i[1], x[1], h)
```

```
91
92  def dy_phi_2d(i, x, h):
93      return phi(i[0], x[0], h) * grad_phi(i[1], x[1], h)
94
95  def dx_hat_2d(i, x, h):
96      return grad_hat(i[0], x[0], h) * hat(i[1], x[1], h)
97
98  def dy_hat_2d(i, x, h):
99      return hat(i[0], x[0], h) * grad_hat(i[1], x[1], h)
100
101
102 """-------------------- QUADRATIC BASIS FUNCTIONS ----------------------"""
103
104 def quad_o(i, x, h):
105     """function to generate even components of the quadratic basis
106
107     Args:
108         i (_float_): position of peak
109         x (_float_): variable
110         h (_type_): width
111
112     Returns:
113         float:
114     """
115     return np.piecewise(x, [x <= (i-h), (x > (i-h))&(x <= i+h - epsilon), x>=(i+h -
    epsilon)],
116                         [0          , lambda x: 1 - ((x-i)/h)**2      , 0
          ])
117
118 def quad_e(i, x, h):
119     """function to generate odd components of quadratic basis
120
121     Args:
122         i (_float_): position of peak
123         x (_float_): variable
124         h (_type_): width
125
126     Returns:
127         float:
128     """
129     return np.piecewise(x, [x <= (i-(2*h)), (x > (i-(2*h))) & (x <= i)
            , (x > i) & (x < (i+(2*h)))                          , x>=(i+(2*h))],
130                         [0              , lambda x: (1/(2*(h**2)))*(x-(i-(2*h)))*(x
    -(i-h)) , lambda x: (1/(2*(h**2)))*(x-(i+(2*h)))*(x-(i+h)) , 0           ])
131
132 def quad(i, x, h):
133     """function for vertices of quadratic basis elements
134
135     Args:
136         i (_float_): position of peak
137         x (_float_): variable
138         h (_type_): width
139
140     Returns:
141         float:
142     """
143
144     if (round(i/h) % 2) == 1:
145         return quad_o(i, x, h)
146     else:
147         return quad_e(i, x, h)
148
149 def quad_2d(i, x, h):
150     """2 dimensional quadratic basis function
151
```

```
    Args:
        i (_float_): position of peak
        x (_float_): variable
        h (_type_): width

    Returns:
        float:
    """
    return quad(i[0], x[0], h) * quad(i[1], x[1], h)


def psi(i, x, h, boundary=[0, 1]):
    """Piecewise Quadratic basis fuction with peak at i and element of size 2*h

    Args:
        i (float): peak of function
        x (float): variable
        h (float): half the width of element
        boundary (list, optional): boundary of domain. Defaults to [0, 1].

    Returns:
        float:
    """
    if (i == np.array(boundary)).any():
        return 0*x
    else:
        return quad(i, x, h)

def grad_psi(i, x, h, boundary=[0, 1]):
    """Piecewise quadratic basis function with peak at i and element of size 2*h

    Args:
        i (float): peak of function
        x (float): variable
        h (float): half the width of element
        boundary (list, optional): boundary of domain. Defaults to [0, 1].

    Returns:
        float:
    """
    if (i == np.array(boundary)).any():
        return 0*x
    else:
        return np.gradient(quad(i, x, h), h)


def psi_2d(i, x, h):
    """Piecewise Quadratic 2 dimensional basis function with peak at i and element of
    size (2*h)**2

    Args:
        i (float): peak of function
        x (float): variable
        h (float): half the width of element

    Returns:
        float:
    """
    return psi(i[0], x[0], h) * psi(i[1], x[1], h)

def dx_psi_2d(i, x, h):
    """the derivative with respect to x of a 2d quadratic basis function

    Args:
        i (float): peak of function
```

```
        x (float): variable
        h (float): half the width of element

    Returns:
        float:
    """

    return grad_psi(i[0], x[0], h) * psi(i[1], x[1], h)

def dy_psi_2d(i, x, h):
    """the derivative with respect to y of a 2d quadratic basis function

    Args:
        i (float): peak of function
        x (float): variable
        h (float): half the width of element

    Returns:
        float:
    """

    return psi(i[0], x[0], h) * grad_psi(i[1], x[1], h)


"""-------------------- EXTRA FUNCTIONS -----------------------"""


def conv_sol(solution, x, basis_func, vertices, h):
    """Converts a solution vector (an array) into an interpolated function on x with corresponding
    basis functions (basis_func).

    Args:
        solution (numpy array): solution vector u to matrix equation Au=F
        x (numpy array): domain over which we want this new function over
        basis_func (function): one of the basis functions. The function is a variable here so do not
        put in any variables into it.
        vertices (list): list of vertices which correspond to each entry of the solution vector u
        h (float): width of element. This is the same h as you would have used in defining your mesh

    Returns:
        numpy array:
    """

    u = 0
    for i, vert in enumerate(vertices):
        u = u + (solution[i] * basis_func(vert, x, h))
    return u
```

Listing 14: basis_functions.py

```
"""
Defining norms on functions to find error in calcualtions
"""

import numpy as np

def l_squ_norm(f, x0):
    """The l squared norm for functions of one variable, or here for 1 dimensional arrays.

    Args:
```

```
        f (numpy array): function given by numpy array
        x0 (numpy array): array on which the function is defined on

    Returns:
        float:
    """
    return np.trapz( np.abs(f)**2 , x0, axis=0)

def l_squ_norm_2d(f, x0):
    """The l squared norm for functions of two variable, or here for 2 dimensional
    arrays.

    Args:
        f (numpy array): function given by numpy array
        x0 (numpy array): tuple of arrays on which the function is defined on

    Returns:
        float:
    """

    return np.sqrt( np.trapz(np.trapz( np.abs(f)**2 , x0[0], axis=0), x0[1], axis=0) )
```

Listing 15: norms.py

```
import numpy as np
#import basis_functions as bf

import os
import sys
current_dir = os.getcwd()
top_dir = os.path.abspath(os.path.join(current_dir, ".."))
sys.path.append(top_dir)

from tools import basis_functions as bf

"""------------------------------------ APPROXIMATE VECTOR PRODUCTS
    -------------------------------------------------"""
# this section contains functions which will predominantly use the numpy gradient
    function to calculate derivatives

def grad_dot_grad(func0, func1, h):
    """returns a dot product of gradients of two functions of two variables.
    ie, grad(func0) dot grad(func1)

    Args:
        func0 (function): a numpy function of two variables
        func1 (function): a numpy function of two variables

    Returns:
        numpy array:
    """

    dy_f0, dx_f0 = np.gradient(func0, h)
    dy_f1, dx_f1 = np.gradient(func1, h)

    return np.multiply(dx_f0, dx_f1) + np.multiply(dy_f0, dy_f1)

def func_mul_pdfunc(func0, func1, h, axis):
    """returns a product of a function and a partial derivative of another function.
    ie, func0 * partial_derivative(func1)

    Args:
        func0 (function): a numpy function of two variables
        func1 (function): a numpy function of two variables
        axis (0 or 1): pick 1 for partial derivative with respect to x or 0 for
```

```
    partial derivative with respect to y

    Returns:
        numpy array:
    """
    pd_f1 = np.gradient(func1, h, axis=axis)

    return np.multiply(pd_f1, func0)


"""---------------------------------------- PRECISE VECTOR PRODUCTS
    ----------------------------------------------"""
# this section contains precise evaluations of important vector products using pen and
     paper maths as much as possible

def grad_dot_grad_phi2d(vert0, vert1, X, h):
    """returns a dot product of gradients of two bf.phi_2d functions.
    mathematically --> grad(phi2d) dot grad(phi2d)

    Args:
        vert0 (float): peak of associated basis function
        vert1 (float): peak of associated basis function
        X (numpy array): array to evaluate functions on
        h (float): width between nodes

    Returns:
        numpy array:
    """

    x, y = X
    i, j = vert0
    a, b = vert1

    dx_0 = bf.grad_phi(i, x, h) * bf.phi(j, y, h)
    dy_0 = bf.phi(i, x, h)      * bf.grad_phi(j, y, h)

    dx_1 = bf.grad_phi(a, x, h) * bf.phi(b, y, h)
    dy_1 = bf.phi(a, x, h)      * bf.grad_phi(b, y, h)

    return dx_0*dx_1 + dy_0*dy_1

def grad_dot_grad_psi2d(vert0, vert1, X, h):
    """returns a dot product of gradients of two bf.psi_2d functions.
    mathematically --> grad(psi2d) dot grad(psi2d)

    Args:
        vert0 (float): peak of associated basis function
        vert1 (float): peak of associated basis function
        X (numpy array): array to evaluate functions on
        h (float): width between nodes

    Returns:
        numpy array:
    """

    x, y = X
    i, j = vert0
    a, b = vert1

    dx_0 = bf.grad_psi(i, x, h) * bf.psi(j, y, h)
    dy_0 = bf.psi(i, x, h)      * bf.grad_psi(j, y, h)

    dx_1 = bf.grad_psi(a, x, h) * bf.psi(b, y, h)
    dy_1 = bf.psi(a, x, h)      * bf.grad_psi(b, y, h)

```

```
    return dx_0*dx_1 + dy_0*dy_1

def grad_dot_grad_hat2d(vert0, vert1, X, h):
    """returns a dot product of gradients of two bf.hat_2d functions.
    mathematically --> grad(psi2d) dot grad(psi2d)

     Args:
        vert0 (float): peak of associated basis function
        vert1 (float): peak of associated basis function
        X (numpy array): array to evaluate functions on
        h (float): width between nodes

    Returns:
        numpy array:
    """

    x, y = X
    i, j = vert0
    a, b = vert1

    dx_0 = bf.grad_hat(i, x, h) * bf.hat(j, y, h)
    dy_0 = bf.hat(i, x, h)      * bf.grad_hat(j, y, h)

    dx_1 = bf.grad_hat(a, x, h) * bf.hat(b, y, h)
    dy_1 = bf.hat(a, x, h)      * bf.grad_hat(b, y, h)

    return (dx_0*dx_1) + (dy_0*dy_1)

def gdg_phi_and_hat_2d(vert0, vert1, X, h, hat_bdry=[]):
    """returns a dot product of gradients of two bf.hat_2d functions.
    mathematically --> grad(phi2d) dot grad(psi2d)

     Args:
        vert0 (float): peak of associated basis function
        vert1 (float): peak of associated basis function
        X (numpy array): array to evaluate functions on
        h (float): width between nodes
        hat_bdry (list): location of boundary for hat functions

    Returns:
        numpy array:
    """

    x, y = X
    i, j = vert0
    a, b = vert1

    dx_0 = bf.grad_phi(i, x, h) * bf.phi(j, y, h, hat_bdry)
    dy_0 = bf.phi(i, x, h)      * bf.grad_phi(j, y, h, hat_bdry)

    dx_1 = bf.grad_phi(a, x, h) * bf.phi(b, y, h, hat_bdry)
    dy_1 = bf.phi(a, x, h)      * bf.grad_phi(b, y, h, hat_bdry)

    return (dx_0*dx_1) + (dy_0*dy_1)
```

Listing 16: vector_products.py