

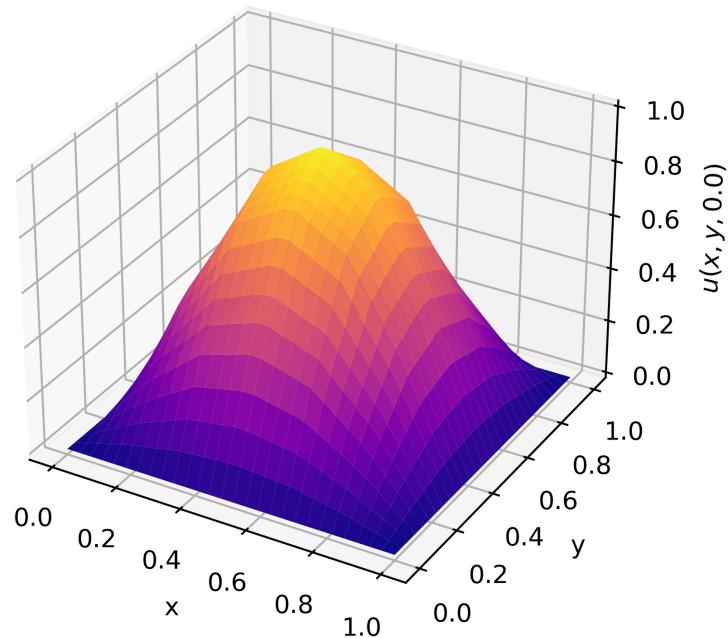
---

# The Finite Element Method

applied to Selected Differential Equations

---

by Patryk Drozd and Adriana Voloshyna



Computational Physics Project  
Maynooth University  
2024

# Contents

<b>1</b>	<b>Background</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
2.1	Computational Methods . . . . .	2
2.2	Background of the Finite Element Method . . . . .	3
<b>3</b>	<b>One Dimensional Poisson Problem</b>	<b>3</b>
3.1	Derivation of the Weak Formulation . . . . .	3
3.2	Discretisation of the Function Space . . . . .	4
3.3	Basis Functions . . . . .	5
3.4	Discretisation of the Weak Formulation . . . . .	5
3.5	Linear System Representation . . . . .	6
3.6	Matrix $A$ . . . . .	6
3.7	Matrix $F$ . . . . .	6
3.8	Numerical Implementation and Solution . . . . .	7
<b>4</b>	<b>Two Dimensional Poisson Problem</b>	<b>8</b>
4.1	The Weak Formulation . . . . .	9
4.2	Basis Functions and Discretisation . . . . .	9
4.3	Implementation and Results . . . . .	11
4.4	Another Two Dimensional Poisson Problem . . . . .	12
4.5	A note on Error Analysis . . . . .	13
<b>5</b>	<b>Heat Equation</b>	<b>13</b>
5.1	Discretisation of Time . . . . .	14
5.2	The Weak Formulation and Spacial Discretisation . . . . .	14
5.3	Implementation and Results . . . . .	15
<b>6</b>	<b>Stokes Equations</b>	<b>16</b>
6.1	The Weak Formulation . . . . .	16
6.2	Basis Functions . . . . .	17
6.3	Discretisation of the Weak Formulation . . . . .	18
6.4	Matrix Formulation . . . . .	20
6.5	Implementation and Results . . . . .	21
6.6	Error Analysis . . . . .	23
<b>7</b>	<b>Conclusion</b>	<b>24</b>
<b>8</b>	<b>Further Reading</b>	<b>25</b>
<b>9</b>	<b>Acknowledgements</b>	<b>25</b>
<b>10</b>	<b>References</b>	<b>25</b>
<b>11</b>	<b>Code</b>	<b>26</b>

# 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

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

with the boundary condition

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

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

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

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

$$V := \{v \mid v \text{ is continuous on } [0,1], v' \text{ is piecewise continuous and bounded on } [0,1], v(0) = v(1) = 0\}$$

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

$$(u', \phi') = (f, \phi) \quad \forall \phi \in V$$

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  $\phi$  and integrate over our domain to obtain

$$-\int_0^1 u''(x)\phi(x) dx = \int_0^1 f(x)\phi(x) dx \quad \forall \phi \in V$$

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

$$\begin{aligned} -([u'(x)\phi(x)]_0^1 + \int_0^1 u'(x)\phi'(x) dx) = \\ \int_0^1 u'(x)\phi'(x) dx - [u'(1)\phi(1) - u'(0)\phi(0)] = \int_0^1 f(x)\phi(x) dx \quad \forall \phi \in V. \end{aligned}$$

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

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

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

$$(u', \phi') = (f, \phi) \quad \forall \phi \in V$$

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

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

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  $\forall \phi \in V$ , we can conclude that

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

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

### 3.2 Discretisation of 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  $x_0 = 0, \dots, x_n = 1$ .

On the interval between each nodal point  $(x_i, x_{i+1})$ , we can define a **Finite Element**  $(K, P, \Sigma)$  where

- K is a cell of the mesh, also called an element
- P are polynomials on K and
- $\Sigma$  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,

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

### 3.3 Basis Functions

The basis functions  $\phi_i$  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.

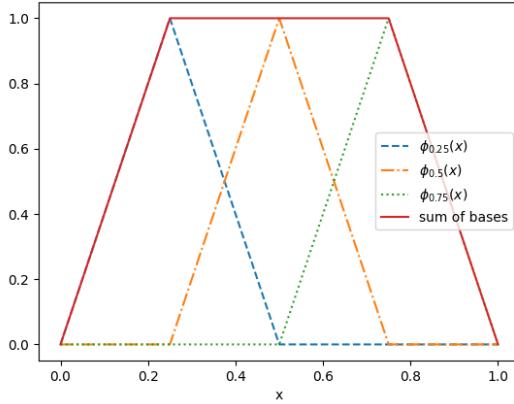


Figure 1: Sum of linear bases

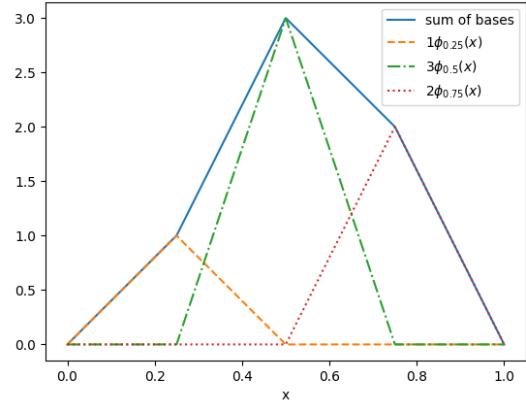


Figure 2: Sum of scaled linear bases

### 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  $u_h \in V_h$  such that

$$(u'_h, \phi'_h) = (f, \phi_h) \quad \forall \phi_h \in V_h.$$

We can write  $u'_h$  in terms of its basis functions to obtain:

$$((\sum_{j=0}^n u_j \phi_j)', \phi'_h) = (f, \phi_h) \quad \forall \phi_h \in V_h.$$

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

$$\begin{aligned} ((\sum_{j=0}^n u_j \phi_j)', \phi'_i) &= (f, \phi_i) \quad \forall 0 \leq i \leq n \\ \iff \sum_{j=0}^n u_j (\phi'_j, \phi'_i) &= (f, \phi_i) \quad \forall 0 \leq i \leq n. \end{aligned}$$

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  $\vec{u} \in \mathbb{R}^n$  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  $(\phi'_i, \phi'_i)$ . Lets take a look at one basis function  $\phi_i$  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

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

By symmetry,

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

Then

$$\begin{aligned} \int_0^1 \phi'_i(x) \phi'_i(x) dx &= \int_{x_{i-1}}^{x_{i+1}} \phi'_i(x) \phi'_i(x) dx = \int_{x_{i-1}}^{x_i} \left(\frac{1}{h}\right) \left(\frac{1}{h}\right) dx + \int_{x_i}^{x_{i+1}} \left(-\frac{1}{h}\right) \left(-\frac{1}{h}\right) dx \\ &= \left[\frac{x}{h^2}\right]_{x_{i-1}}^{x_i} + \left[\frac{x}{h^2}\right]_{x_i}^{x_{i+1}} = \frac{h}{h^2} + \frac{h}{h^2} = \frac{2}{h}. \end{aligned}$$

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.

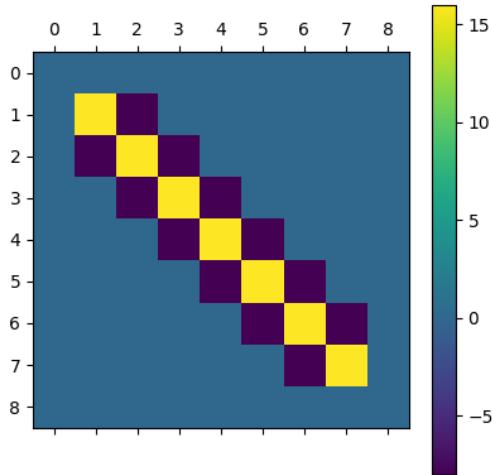


Figure 3: Matrix  $A$  for 9 nodal points

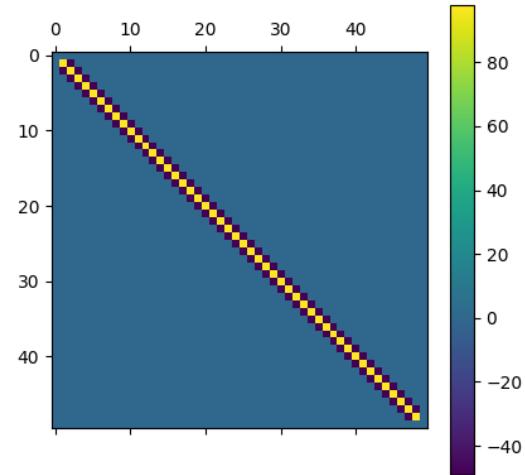


Figure 4: Matrix  $A$  for 50 nodal points

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  $\phi_i$  for all  $i$ . Then, for each basis function  $\phi_i$ , we take the scalar product of  $\phi_i$  and  $\phi_j$ , 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 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

$$\begin{aligned} u(x) &= \frac{1}{2}x(x-1) && \text{for } f(x) = 1, \\ u(x) &= \frac{1}{6}x(x^2-1) && \text{for } f(x) = x. \end{aligned}$$

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.

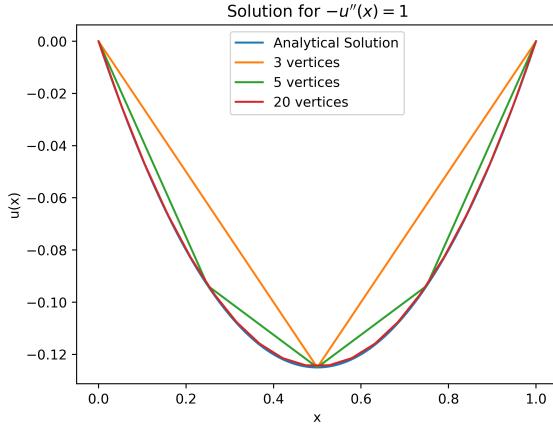


Figure 5: Solutions for various mesh sizes

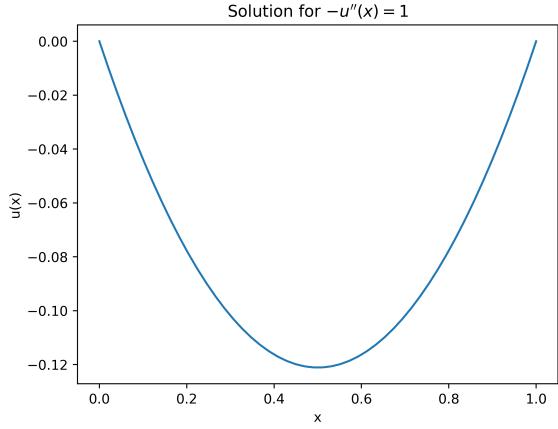


Figure 6: Solution for 50 nodal points

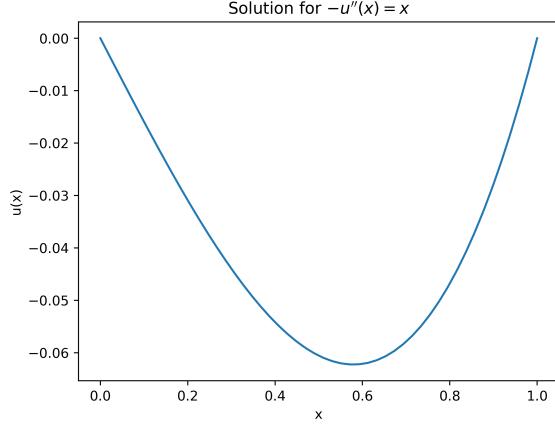


Figure 7: Solution for 50 nodal points

## 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

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

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:

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

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

$$\int_{\Omega} \nabla^2 u \phi \, dx dy = - \int_{\Omega} \nabla u \cdot \nabla \phi \, dx dy + \int_{\partial\Omega} \nabla u \cdot \hat{n} \phi \, dx dy = \int_{\Omega} f(x, y) \phi(x, y) \, dx dy \quad \forall \phi \in V.$$

Implementing the Neumann boundary conditions gives

$$\int_{\Omega} \nabla u \cdot \nabla \phi \, dx dy = - \int_{\Omega} f \phi \, dx dy - \int_{\partial\Omega_N} x(x-1) \phi \, dx dy \quad \forall \phi \in V.$$

## 4.2 Basis Functions and Discretisation

We once again only consider a discretised subspace of  $V$ , namely  $V_h$  where we restrict each  $v_h \in V_h$  to be polynomial on each element of our mesh. We similarly define basis functions  $\phi_i$  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

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

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

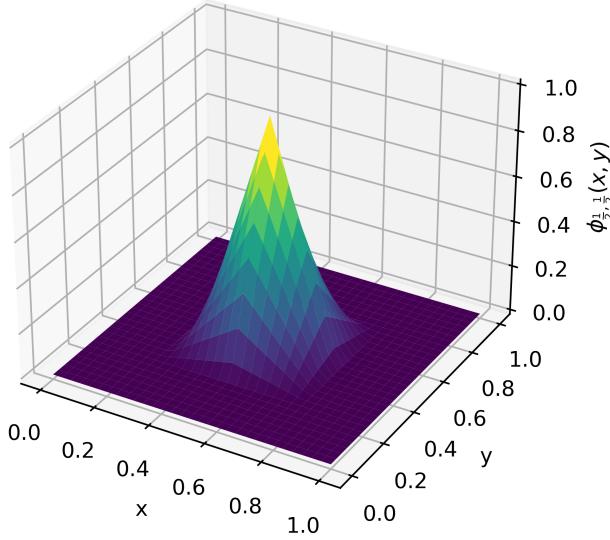


Figure 8: Two dimensional basis function with peak at  $(x, y) = (0.5, 0.5)$  and distance between nodes  $h = 0.25$

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  $\phi$  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

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

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

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

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  $\Omega$ , 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  $7 \times 7$  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.

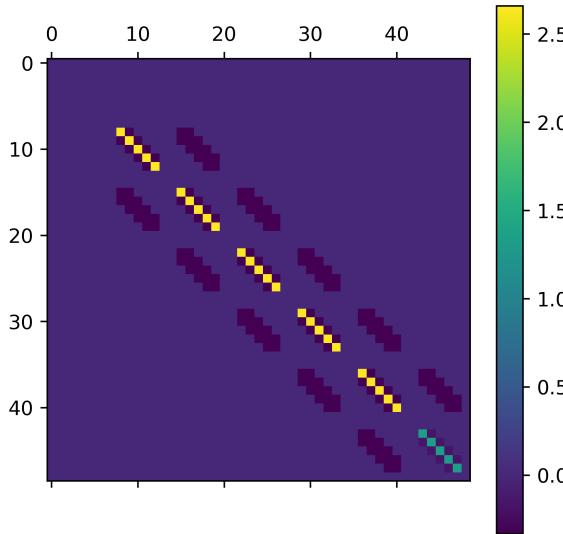


Figure 9: Matrix  $A$  for 49 nodal points (with Dirichlet and Neumann BCs)

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

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

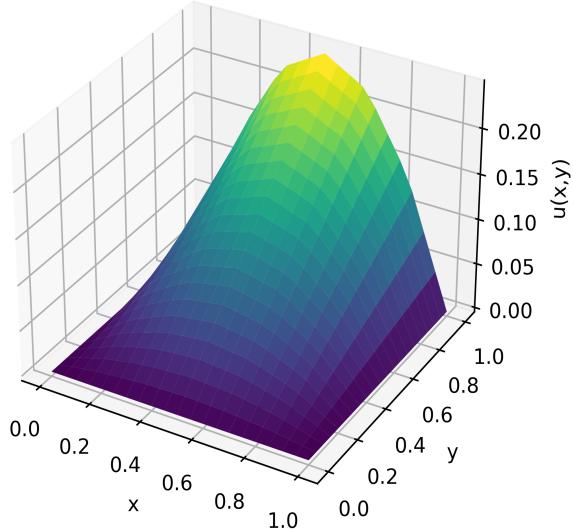


Figure 10: Solution to  $\nabla^2 u = -2y$  with mixed boundary conditions

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

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

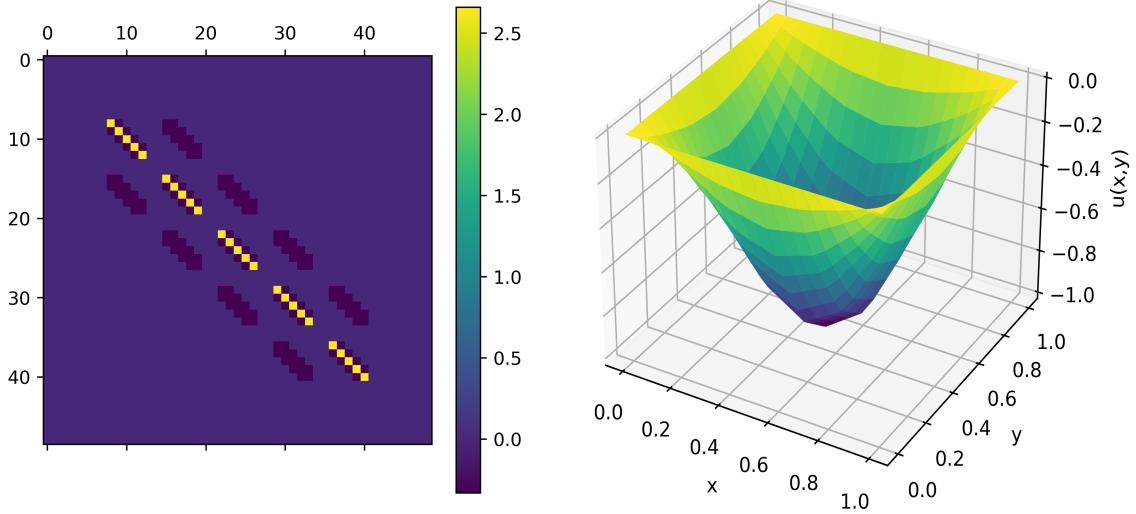


Figure 11: Matrix  $A$  for 49 nodal points

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  $5 \times 5$  grid, the error of our solution was of magnitude  $\approx 0.030655$ . However if we increase the size of the grid to  $7 \times 7$ , the error decreased to  $\approx 0.014620$ .

## 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  $\Omega$  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  $\Delta t$  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

$$\begin{aligned} u^0 &= \sin(\pi x) \sin(\pi y) \\ u^{s+1} - \nabla^2 u^{s+1} \Delta t &= u^s + f \Delta t \quad \text{for } s = 0, 1, 2, \dots \end{aligned}$$

## 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  $\phi$  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:

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

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  $u^{s+1}$ .

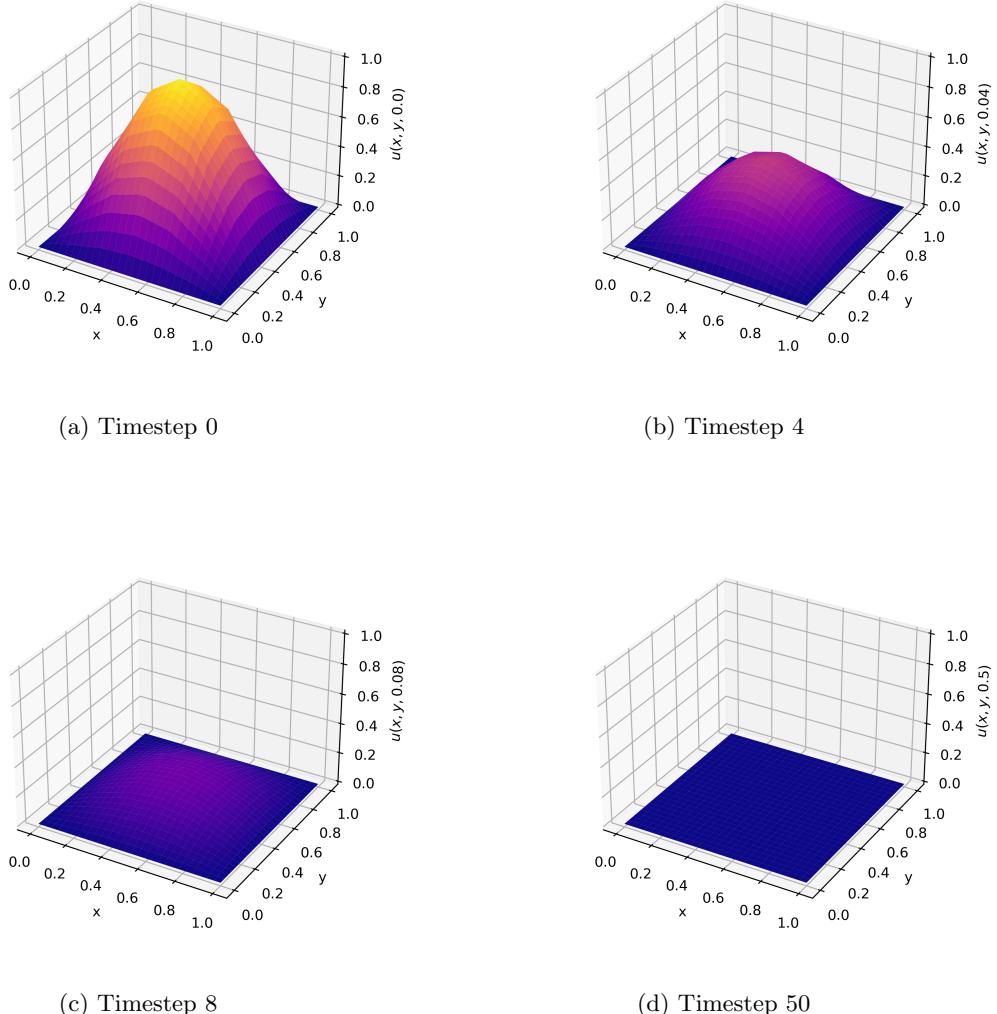


Figure 13: Snapshots of the solution to  $\frac{\partial u}{\partial t} = \nabla^2 u$  with Dirichlet boundary conditions and initial conditions  $u(x, y, 0) = \sin(\pi x) \sin(\pi y)$

### 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  $\Delta t$ . 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

- $\mu$  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  $\mu$ ) we aim to find a suitable  $\vec{u}$  and  $p$  which satisfy the equations above. We let our domain  $\Omega$  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:

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

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  $\phi_1, \phi_2, \phi_3$  the weak formulations of each equation can be expressed as:

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

## 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.

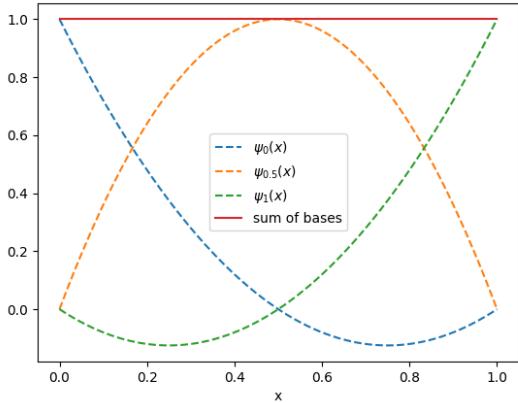


Figure 14: Sum of quadratic bases

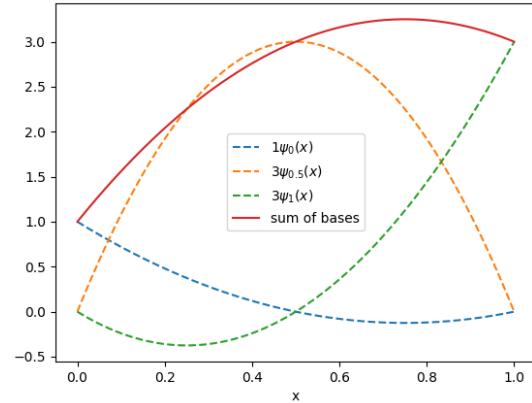


Figure 15: Sum of scaled quadratic bases

To gain an 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 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

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

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

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

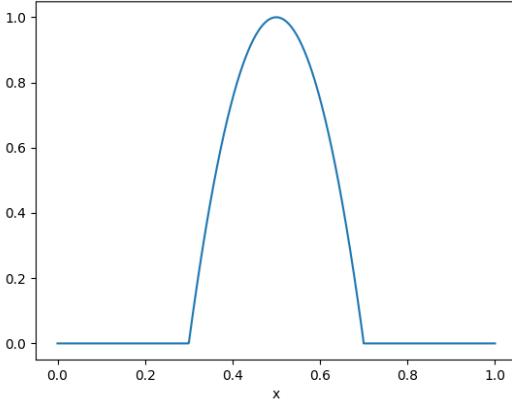


Figure 16: Odd Quadratic Basis Function

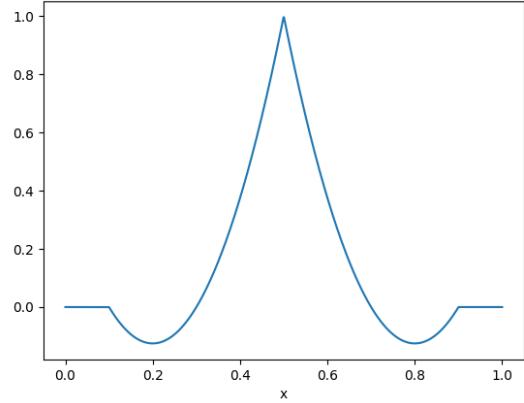


Figure 17: Even Quadratic Basis Function

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

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

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

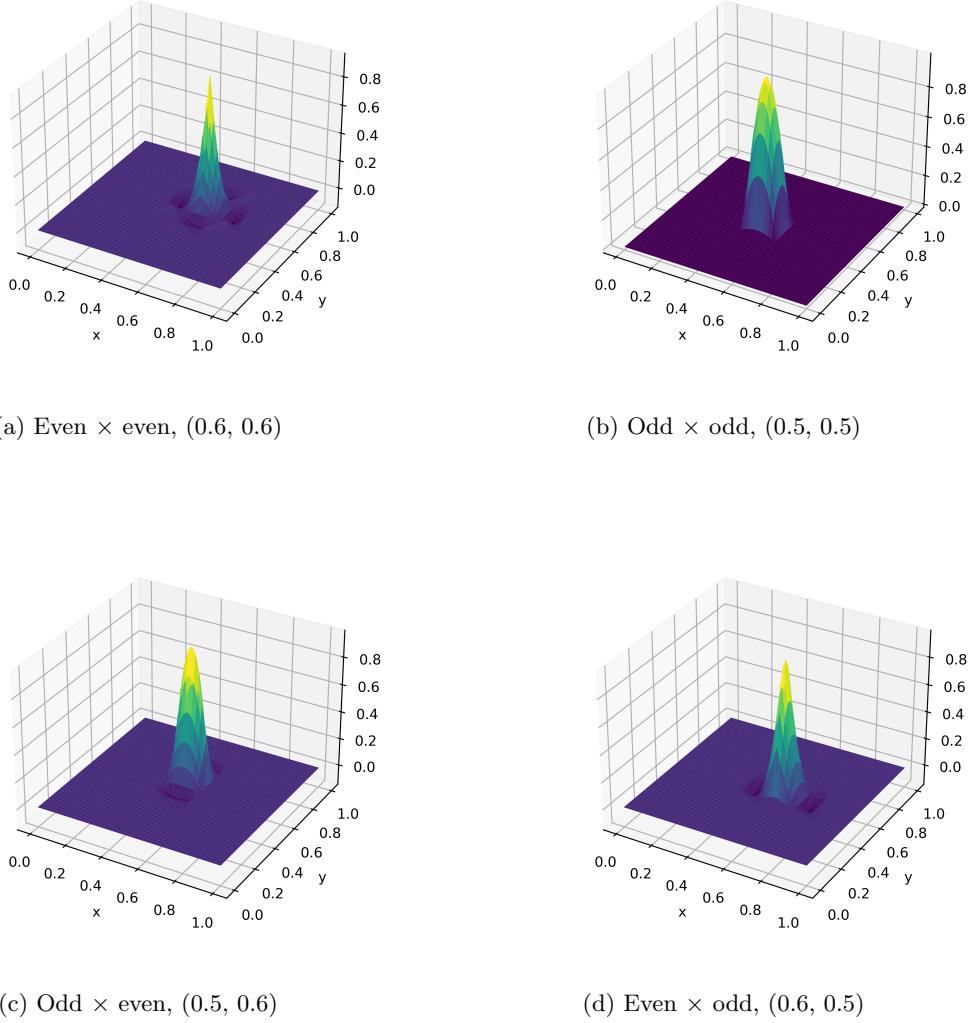


Figure 18: Two dimensional quadratic basis function

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

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

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

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

$$\sum_{i=0}^n u_i^1 \int_{\Omega} \frac{\partial \psi_j}{\partial x} \phi_i \, dx dy + \sum_{i=0}^n u_i^2 \int_{\Omega} \frac{\partial \psi_j}{\partial y} \phi_i \, dx dy = 0 \quad (3)$$

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 ( $3 \times 3$ ) main parts. We first label some of the integral terms in the above equations for clarity.

$$\begin{aligned} a_{ij} &:= \mu \int_{\Omega} \nabla \psi_i \cdot \nabla \psi_j \, dx dy \\ b_{ij} &:= - \int_{\Omega} \phi_i \frac{\partial \psi_j}{\partial x} \, dx dy \\ c_{ij} &:= - \int_{\Omega} \phi_i \frac{\partial \psi_j}{\partial y} \, dx dy \\ d_j^1 &:= \int_{\Omega} f_1 \psi_j \, dx dy \quad d_j^2 := \int_{\Omega} f_2 \psi_j \, dx dy \end{aligned}$$

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  $F$ :

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

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

$$\begin{aligned} u_1 &= 2\pi(1 - \cos(2\pi x)) \sin(2\pi y) \\ u_2 &= -2\pi(1 - \cos(2\pi y)) \sin(2\pi x) \\ p &= \sin(2\pi x) \sin(2\pi y). \end{aligned}$$

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.

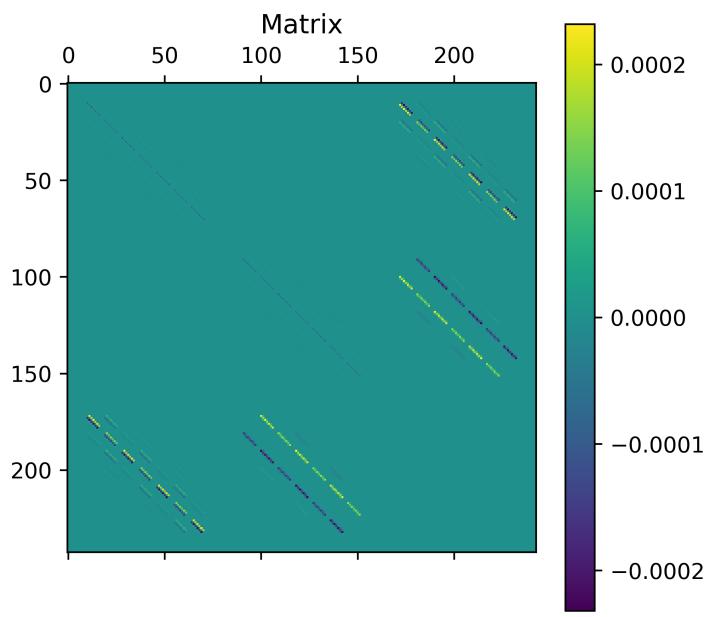


Figure 19: Matrix  $A$  for 81 nodes

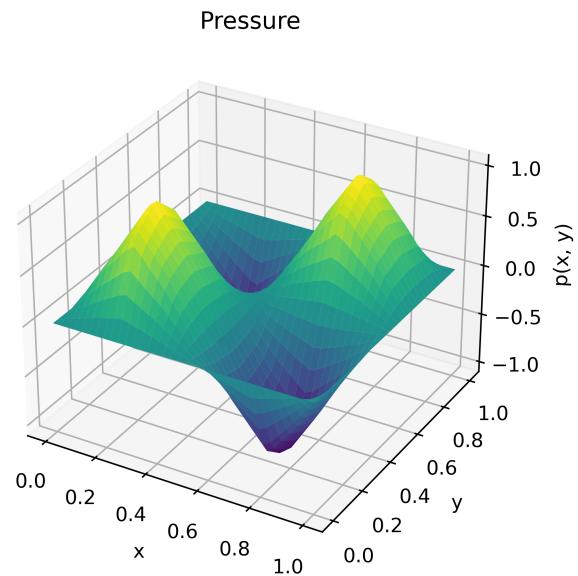


Figure 20: Numerically derived pressure function  $p$

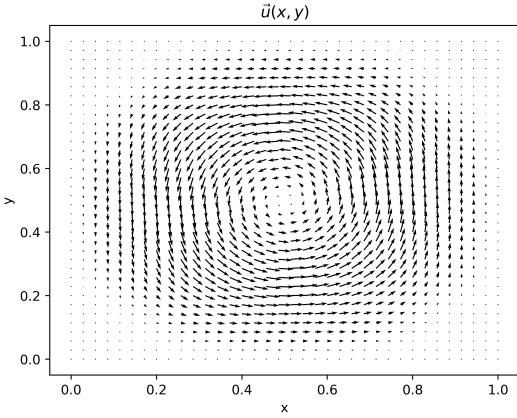


Figure 21: Vector plot of  $\vec{u}$

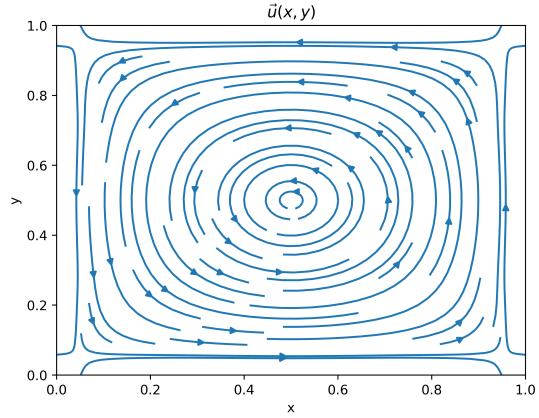


Figure 22: Stream plot of  $\vec{u}$

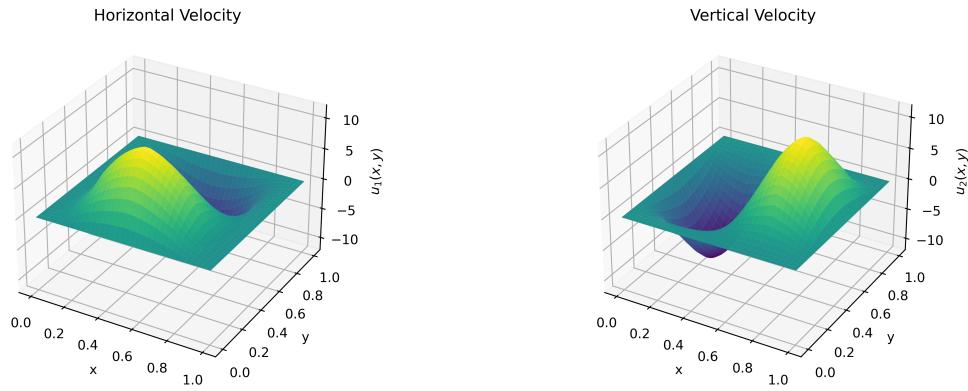


Figure 23: Horizontal velocity  $u_1$

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  $7 \times 7$  grid, we obtain an error of:

- $\approx 0.864872$  for  $u_1$
- $\approx 0.886772$  for  $u_2$
- $\approx 0.087308$  for  $p$

On a  $9 \times 9$  grid, the errors we obtain for our solutions were:

- $\approx 0.717391$  for  $u_1$
- $\approx 0.717148$  for  $u_2$
- $\approx 0.060192$  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/~jb Burkardt/classes/fem2011/femstokes.pdf

```

## 11 Code

```

1 """
2 This script solves the 1 dimensional poisson problem with dirichlet boundary
3 conditions.
4 with solution
5 -----
6 ----- in LaTeX code -----
7
8 $u(x) = \frac{1}{2} x(x-1)
9
10 and forcing function
11
12 $f(x) = 1
13
14 """
15
16 import numpy as np
17 import matplotlib.pyplot as plt
18
19 import os
20 import sys
21 current_dir = os.getcwd()
22 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
23 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
33     easy way to
34     run solver with various parameters.
35
36     Args:
37         H (int): number of nodes
38
39     Returns:
40         tuple: solution to equation, solution matrix A, solution matrix F, error
41     """
42
43     vertices = list(np.linspace(0, 1, H))
44     elements = [[vertices[i], vertices[i+1]] for i in range(H-1)]
45     h = vertices[1] - vertices[0]
46
47     # defining basis functions and their derivatives. This was done initially to swap
48     # out linear basis functions to quadratic basis function just because we can
49     def basis_func(vert, x):
50         return bf.phi(vert, x, h)
51
52     def d_basis_func(vert, x):
53         return bf.grad_phi(vert, x, h)
54
55     def force(x):
56         return -1
57
58     # defining matrix to calculate entries for
59     A = np.zeros((H, H))
60     F = np.zeros((H,))
61
62     # functions for calculating entries of matrices
63     def a(vert0, vert1, x):
64         return d_basis_func(vert0, x) * d_basis_func(vert1, x)
65
66     def f(vert0, x):
67         return basis_func(vert0, x) * force(x)
68
69     # main bit of code iterating over elements of mesh (here a 1dim line) and
70     # calcualting entries of the matrices
71     for ele in elements:
72         x = np.linspace(ele[0], ele[1], steps)
73
74         for vert0 in ele:
75             j = vertices.index(vert0)
76             F[j] = F[j] + np.trapz(f(vert0, x), x)
77
78             for vert1 in ele:
79                 i = vertices.index(vert1)
80                 A[i, j] = A[i, j] + np.trapz(a(vert0, vert1, x), x)
81
82     # final calculation of solution
83     A_inverted = np.linalg.pinv(A)
84     solution = np.matmul(A_inverted, F)
85
86     vertices = np.array(vertices)
87     error = nrm.l_sq_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
118 functions.
119 There is an  $L^2$  norm which is used to compare functions. We would simply take the
120 difference
121 of our solution and its analytical solution and throw it into this norm.
122 """
123 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 boundary 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, "..", ".."))

```

```

24 sys.path.append(top_dir)
25
26 from tools import basis_functions as bf
27 from tools import vector_products as vp
28 from tools import norms as nrm
29
30 # main parameters to change
31 steps = 1000 # accuracy or steps in integrals, make this smaller for significantly
   quicker run time
32 H = 5 # $H^2$ is the number of nodes in the mesh, tweak this parameter to get a denser
   mesh
33 # important to increase steps along side H
34
35 # defining mesh
36 x = np.linspace(0, 1, H)
37 y = np.linspace(0, 1, H)
38 h = x[1] - x[0]
39 vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()
40 elements = [[[x[i], y[j]],
41               [x[i+1], y[j]],
42               [x[i], y[j+1]],
43               [x[i+1], y[j+1]]] for i in range(H-1) for j in range(H-1)]
44
45
46 # defining functions for integrands
47 def force(x):
48     return -2*(np.pi**2)*np.sin(np.pi*x[0])*np.sin(np.pi*x[1])
49
50 def f(vert, x):
51     return bf.phi_2d(vert, x, h) * force(x)
52
53 def a(vert0, vert1, x):
54     return vp.grad_dot_grad_phi2d(vert0, vert1, x, h)
55
56 A = np.zeros((H**2, H**2))
57 F = np.zeros((H**2))
58
59 t0 = time.time()
60 for k, ele in enumerate(elements):
61     x0 = np.linspace(ele[0][0], ele[3][0], steps)
62     y0 = np.linspace(ele[0][1], ele[3][1], steps)
63     X0, Y0 = np.meshgrid(x0, y0)
64
65     percentage = round(100 * ((k)/(len(elements)-1)), 1)
66     print(f"{percentage}%", end="\r")
67
68     for vert0 in ele:
69         j = vertices.index(vert0)
70         F[j] = F[j] + np.trapz(np.trapz(f(vert0, (X0, Y0)), y0, axis=0), x0, axis=0)
71
72         for vert1 in ele:
73             i = vertices.index(vert1)
74             A[j, i] = A[j, i] + np.trapz(np.trapz(a(vert0, vert1, (X0, Y0)), y0,
75             axis=0), x0, axis=0)
76
77 t1 = time.time()
78 print(f"time taken: {(t1-t0)/60} minutes")
79
80 # this is solving the matrix equation Au = F
81 solution = np.matmul(np.linalg.pinv(A), F)
82
83 # the rest is plotting
84 N = 3*H
85 x0 = np.linspace(0, 1, N)
86 y0 = np.linspace(0, 1, N)

```

```

86 X0, Y0 = np.meshgrid(x0, y0)
87
88 fig = plt.figure()
89 ax = plt.axes(projection ='3d')
90 ax.plot_surface(X0, Y0, bf.conv_sol(solution, (X0, Y0), bf.phi_2d, vertices, h), cmap=
91 "viridis")
92 ax.set_xlabel('x')
93 ax.set_ylabel('y')
94 ax.set_zlabel('u(x,y)')
95 ax.set_title(r"\nabla^2 u(x,y) = -2 \pi^2 \sin(\pi x) \sin(\pi y) ")
96 #plt.savefig(f"(Poisson_2d)_({vertex_num_{H**2}})", dpi=500)
97
98 plt.matshow(A)
99 plt.colorbar()
100 #plt.savefig(f"(Poisson_2d)_({mat_A}_({vertex_num_{H**2}})", dpi=500)
101 plt.show()
102
103 #error check
104 math_sol = lambda x: -np.sin(np.pi*x[0])*np.sin(np.pi*x[1])
105 ele_sol = bf.conv_sol(solution, (X0, Y0), bf.phi_2d, vertices, h)
106 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

```

1 """
2 Script solving the 2d poisson problem with neumann boundary conditions.
3
4 With manufactured forcing function
5
6 ----- in LateX code-----
7
8 $f(x, y) = -2y
9
10 neumann boundary conditions
11
12 $\partial u \cdot \hat{n} = -x(x-1)$
13
14 and solution
15
16 $u(x, y) = -yx(x - 1)
17
18 """
19
20 import numpy as np
21 import matplotlib.pyplot as plt
22 import time
23
24 import os
25 import sys
26 current_dir = os.getcwd()
27 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
28 sys.path.append(top_dir)
29
30 from tools import basis_functions as bf
31 from tools import vector_products as vp
32 from tools import norms as nrm
33
34 # main parameters to change
35 steps = 200 # accuracy or steps in integrals, make this smaller for significantly
36 # quicker run time
37 H = 7 # $H^2$ is the number of nodes in the mesh, tweak this parameter to get a denser
# mesh
38 # defining mesh

```

```

39 x = np.linspace(0, 1, H)
40 y = np.linspace(0, 1, H)
41 h = x[1] - x[0]
42 vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()
43 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)]
44
45
46
47
48 # defining functions for integrands
49 def force(x):
50     x, y = x
51     return -2*y
52
53 def n(x):
54     x, y = x
55     return -x*(x-1)
56
57 def f(vert, x):
58     return bf.phi_and_hat_2d(vert, x, h, [0]) * force(x)
59
60 def a(vert0, vert1, x):
61     return vp.gdg_phi_and_hat_2d(vert0, vert1, x, h, [0])
62
63 def neumann(vert, x):
64     x, y = x
65     int1 = np.trapz(n((x, 0)) * bf.phi_and_hat_2d(vert, (x, 0), h, []), x, axis=0)
66     int2 = np.trapz(n((x, 1)) * bf.phi_and_hat_2d(vert, (x, 1), h, []), x, axis=0)
67     return int1 + int2
68
69 A = np.zeros((H**2, H**2))
70 F = np.zeros((H**2))
71
72
73 t0 = time.time()
74 for k, ele in enumerate(elements):
75     x0 = np.linspace(ele[0][0], ele[3][0], steps)
76     y0 = np.linspace(ele[0][1], ele[3][1], steps)
77     X0, Y0 = np.meshgrid(x0, y0)
78
79     percentage = round(100 * ((k)/(len(elements)-1)), 1)
80     print(f"{percentage}%", end="\r")
81
82     for vert0 in ele:
83         j = vertices.index(vert0)
84         F[j] = F[j] - np.trapz(np.trapz(f(vert0, (X0, Y0)), y0, axis=0), x0, axis=0)
85         + neumann(vert0, (x0, y0))
86
87         for vert1 in ele:
88             i = vertices.index(vert1)
89             A[j, i] = A[j, i] + np.trapz(np.trapz(a(vert0, vert1, (X0, Y0)), y0, axis=0), x0, axis=0)
90
91 t1 = time.time()
92 print(f"time taken: {(t1-t0)/60} minutes")
93
94 # this is solving the matrix equation Au = F
95 solution = np.matmul(np.linalg.pinv(A), F)
96
97 # the rest is plotting
98 N = 3*H
99 x0 = np.linspace(0, 1, N)
100 y0 = np.linspace(0, 1, N)
101 X0, Y0 = np.meshgrid(x0, y0)

```

```

102
103 fig = plt.figure()
104 ax = plt.axes(projection ='3d')
105 ax.plot_surface(X0, Y0, bf.conv_sol(solution, (X0, Y0), bf.hat_2d, vertices, h), cmap=
106     "viridis")
107 ax.set_xlabel('x')
108 ax.set_ylabel('y')
109 ax.set_zlabel('u(x,y)')
110 #ax.set_title(r"\nabla^2 u(x,y) = -2y $ with neumann bdry $n(x,y) = -x(x-1)$")
111 #plt.savefig(f"(Poisson_2d)_({neumann})_(vertex_num_{H**2})", dpi=500)
112
113 plt.matshow(A)
114 plt.colorbar()
115 plt.savefig(f"(Poisson_2d)_({mat})_({neumann})_(vertex_num_{H**2})", dpi=500)
116 plt.show()
117
118 #error check
119 math_sol = lambda x: -np.sin(np.pi*x[0])*np.sin(np.pi*x[1])
120 ele_sol = bf.conv_sol(solution, (X0, Y0), bf.phi_2d, vertices, h)
121 print(f'L squared norm error with {H**2} nodes = {nrm.l_squ_norm_2d(ele_sol - math_sol
122 ((X0, Y0)), (x0, y0))}')

```

Listing 3: poisson\_2dim\_ele.neumann.py

```

1 """
2 This a file is for solving the heat equation with
3 -----
4 ----- in LateX code -----
5
6 $u(x, y, 0) = \sin(\pi x) \sin(\pi y)$
7
8 $f(x, y, t) = 0$"
9
10 -----
11 """
12
13
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import matplotlib.animation as animation
17 import time
18
19 import os
20 import sys
21 current_dir = os.getcwd()
22 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
23 sys.path.append(top_dir)
24
25 from tools import basis_functions as bf
26 from tools import vector_products as vp
27
28 steps = 100 # steps in integrals
29
30 # defining mesh
31 H = 7
32 x = np.linspace(0, 1, H)
33 y = np.linspace(0, 1, H)
34 X, Y = np.meshgrid(x, y)
35 h = x[1] - x[0]
36 vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()
37 elements = [[[x[i], y[j]],
38             [x[i+1], y[j]],
39             [x[i], y[j+1]],
40             [x[i+1], y[j+1]]] for i in range(H-1) for j in range(H-1)]
41

```

```

42 # declaring time stepping necessities
43 dt = .01 # change in time
44 time_steps = 50 # amount of iterations we want
45 u_0 = [np.sin(np.pi*x[0]) * np.sin(np.pi*x[1]) for x in vertices] # initial conditions
46 u_sols = [u_0] # list for keeping track of evolution of system
47 A_sols = []
48
49 def force(x, t):
50     return 0
51
52 # functions for calculating the entries of matrices
53 def f(vert, x, u, time):
54     return (u + (dt * force(x, time))) * bf.phi_2d(vert, x, h)
55
56 def a(vert0, vert1, x):
57     return (bf.phi_2d(vert0, x, h)*bf.phi_2d(vert1, x, h)) + (dt*(vp.
58     grad_dot_grad_phi2d(vert0, vert1, x, h)))
59
60 t0 = time.time()
61 for t in range(time_steps): #time stepping loop
62     A = np.zeros((H**2, H**2))
63     F = np.zeros((H**2))
64
65     # main code, iterating over elements and calculating entries of matrix
66     for k, ele in enumerate(elements):
67         # creating domain of element to iterate over
68         x0 = np.linspace(ele[0][0], ele[3][0], steps)
69         y0 = np.linspace(ele[0][1], ele[3][1], steps)
70         X0, Y0 = np.meshgrid(x0, y0)
71
72         u = bf.conv_sol(u_sols[t], (X0, Y0), bf.phi_2d, vertices, h)
73
74         percentage = round(100 * ((k)/(len(elements)-1)), 1)
75         print(f"time {t} and {percentage}%", end="\r")
76
77         for vert0 in ele:
78             j = vertices.index(vert0)
79             F[j] = F[j] + np.trapz(np.trapz( f(vert0, (X0, Y0), u, t*dt) , y0,
80             , x0, axis=0)
81
82             for vert1 in ele:
83                 i = vertices.index(vert1)
84                 A[j, i] = A[j, i] + np.trapz(np.trapz( a(vert0, vert1, (X0, Y0)) , y0,
85                 , x0, axis=0)
86
87
88
89
90 t1 = time.time()
91 print(f"time taken: {(t1-t0)/60} minutes")
92
93 # creating mesh with higher resolution for plotting
94 N = 4*H
95 x = np.linspace(0, 1, N)
96 y = np.linspace(0, 1, N)
97 X, Y = np.meshgrid(x, y)
98
99 func_u_sols = []
100 for i in range(len(u_sols)):
101     func_u_sols.append(bf.conv_sol(u_sols[i], (X, Y), bf.phi_2d, vertices, h))

```

```

103 """----- PLOTTING -----"""
104
105 # plot and animation
106 len_u_sols = len(func_u_sols)
107 frn = len_u_sols
108 fps = frn//3
109
110 def change_plot(frame_number, func_u_sols, plot):
111     plot[0].remove()
112     plot[0] = ax.plot_surface(X, Y, func_u_sols[frame_number], cmap="plasma", vmin=0,
113                               vmax=1)
113
114 fig = plt.figure()
115 ax = fig.add_subplot(projection='3d')
116 plot = [ax.plot_surface(X, Y, func_u_sols[0])]
117 ax.set_zlim(0, 1)
118 ax.set_xlabel("x")
119 ax.set_ylabel("y")
120 ax.set_zlabel("u(x, y, t)")
121 ax.set_title(r"$\frac{\partial u}{\partial t} = \nabla^2 u $ with $u(x,y,0) = \sin(\pi x)\sin(\pi y)$")
122
123 ani = animation.FuncAnimation(fig, change_plot, frn, fargs=(func_u_sols, plot),
124                               interval=1000 / fps)
124 ani.save('animation.gif', writer='PillowWriter', fps=fps, dpi=400)
125 plt.show()
126
127 # plot of matrix for first iteration
128 plt.matshow(A_sols[0])
129 plt.title("Matrix for first iteration")
130 plt.colorbar()
131 plt.show()
132
133 # creating a sequence of images for report
134 plt.clf()
135 iter1 = 0
136 fig = plt.figure()
137 ax = plt.axes(projection = '3d')
138 ax.axes.set_zlim3d(bottom=0, top=1)
139 ax.plot_surface(X, Y, func_u_sols[iter1], cmap="plasma", vmin=0, vmax=1)
140 ax.set_xlabel('x')
141 ax.set_ylabel('y')
142 ax.set_zlabel(f'$u(x, y, {iter1*dt})$')
143 #plt.savefig(f"heat_equ_iter{iter1}", dpi=400)
144
145 plt.clf()
146 iter2 = int(len_u_sols/12)
147 fig = plt.figure()
148 ax = plt.axes(projection = '3d')
149 ax.axes.set_zlim3d(bottom=0, top=1)
150 ax.plot_surface(X, Y, func_u_sols[iter2], cmap="plasma", vmin=0, vmax=1)
151 ax.set_xlabel('x')
152 ax.set_ylabel('y')
153 ax.set_zlabel(f'$u(x, y, {iter2*dt})$')
154 #plt.savefig(f"heat_equ_iter{iter2}", dpi=400)
155
156 plt.clf()
157 iter3 = int(len_u_sols/6)
158 fig = plt.figure()
159 ax = plt.axes(projection = '3d')
160 ax.axes.set_zlim3d(bottom=0, top=1)
161 ax.plot_surface(X, Y, func_u_sols[iter3], cmap="plasma", vmin=0, vmax=1)
162 ax.set_xlabel('x')
163 ax.set_ylabel('y')
164 ax.set_zlabel(f'$u(x, y, {iter3*dt})$')

```

```

165 #plt.savefig(f"heat_equ_iter{iter3}", dpi=400)
166
167 plt.clf()
168 iter4 = time_steps
169 fig = plt.figure()
170 ax = plt.axes(projection ='3d')
171 ax.axes.set_zlim3d(bottom=0, top=1)
172 ax.plot_surface(X, Y, func_u_sols[iter4], cmap="plasma", vmin=0, vmax=1)
173 ax.set_xlabel('x')
174 ax.set_ylabel('y')
175 ax.set_zlabel(f'${u(x, y, {iter4*dt})}$')
176 #plt.savefig(f"heat_equ_iter{iter4}", dpi=400)

```

Listing 4: heat\_equation\_2+1dim\_ele.py

```

1 """
2 Script for solving the stokes equation with a manufactured forcing functions for the
   solutions.
3
4 ----- in LateX code -----
5
6 $u_1 = 2\pi(1 - \cos(2\pi x)\sin(2\pi y)$
7 $u_2 = -2\pi(1 - \cos(2\pi y)\sin(2\pi x)$
8 $p = \sin(2 \pi x)\sin(2 \pi y)$
9
10 -----
11 """
12 """
13
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import time
17
18 import os
19 import sys
20 current_dir = os.getcwd()
21 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
22 sys.path.append(top_dir)
23
24 from tools import basis_functions as bf
25 from tools import vector_products as vp
26 from tools import norms as nrm
27
28 # main parameters to tweak
29 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
30 steps = 600 # steps in integrals over each element
31 # as you increase H make sure to increase steps
32
33 # defining mesh
34 x = np.linspace(0, 1, H)
35 y = np.linspace(0, 1, H)
36 X, Y = np.meshgrid(x, y)
37 h = x[1] - x[0]
38 vertices = (np.array(np.meshgrid(x, y)).reshape(2, -1).T).tolist()
39
40 # defining quadratic elements
41 quad_elements = []
42 for i in list(np.arange(0, H-1, 2)):
43     for j in list(np.arange(0, H-1, 2)):
44
45         element = []
46         for a in range(0, 3):
47             for b in range(0, 3):
48                 element.append([x[i+a], y[j+b]])

```

```

49         quad_elements.append(element)
50
51
52 # manufacturing solutions to stokes equation
53 mu = 1 # viscosity
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):
    BLOCK = np.zeros((H**2, H**2))
90
91
92     for k, ele in enumerate(quad_elements):
        # creating domain for integration
93         x0 = np.linspace(ele[0][0], ele[-1][0], steps)
94         y0 = np.linspace(ele[0][1], ele[-1][1], steps)
95         X0, Y0 = np.meshgrid(x0, y0)
96
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
103        # main iteration loop
104        for vert0 in ele:
            j = vertices.index(vert0)
            for vert1 in ele:
                i = vertices.index(vert1)
                BLOCK[j, i] = BLOCK[j, i] + np.trapz(np.trapz( a(vert0, vert1, (X0,
    ), h) , y0, axis=0), x0, axis=0)
105
106
107
108

```

```

109     mat_blocks.append(BLOCK)
110
111 # block vector calculations
112 vect_blocks = []
113 force_funcs = [f_1, f_2]
114 for kf, f in enumerate(force_funcs):
115     BLOCK = np.zeros((H**2))
116
117     for k, ele in enumerate(quad_elements):
118         # creating domain for integration
119         x0 = np.linspace(ele[0][0], ele[-1][0], steps)
120         y0 = np.linspace(ele[0][1], ele[-1][1], steps)
121         X0, Y0 = np.meshgrid(x0, y0)
122
123         # keep track of code running
124         percentage = round(100 * ((k)/(len(quad_elements)-1)), 1)
125         print(f"force_{kf} {percentage}%", end="\r")
126
127         # main iteration loop
128         for vert0 in ele:
129             j = vertices.index(vert0)
130             BLOCK[j] = BLOCK[j] + np.trapz(np.trapz((bf.hat_2d(vert0, (X0, Y0), h)) *
131                                         f((X0, Y0)), y0, axis=0), x0, axis=0)
132
133     vect_blocks.append(BLOCK)
134
135 t1 = time.time()
136 print(f"time taken: {round((t1-t0)/60, 2)} minutes")
137
138 # assembly of blocks
139 zero_mat = np.zeros((H**2, H**2))
140 zero_vect = np.zeros((H**2))
141
142 reshaped_mat_blocks = [[mat_blocks[0], zero_mat, mat_blocks[1]],
143                         [zero_mat, mat_blocks[0], mat_blocks[2]],
144                         [mat_blocks[1].T, mat_blocks[2].T, zero_mat]]
145
146 reshaped_vect_blocks = [vect_blocks[0], vect_blocks[1], zero_vect]
147
148 A = np.block(reshaped_mat_blocks)
149 F = np.block(reshaped_vect_blocks)
150
151 # calculation of solution
152 solution = np.matmul(np.linalg.pinv(A), F)
153
154 # creating mesh with higher resolution for plotting
155 N = 4*H
156 x = np.linspace(0, 1, N)
157 y = np.linspace(0, 1, N)
158 X, Y = np.meshgrid(x, y)
159
160 # dissassembling solution vector
161 u1_sol = bf.conv_sol(solution[0 : H**2], (X, Y), bf.psi_2d, vertices, h)
162 u2_sol = bf.conv_sol(solution[H**2 : 2*(H**2)], (X, Y), bf.psi_2d, vertices, h)
163 p_sol = bf.conv_sol(solution[2*(H**2) : ], (X, Y), bf.hat_2d, vertices, h)
164
165 # plotting
166 plt.matshow(A)
167 plt.colorbar()
168 plt.title("Matrix")
169 plt.savefig("Vector_pde_matrix", dpi=500)
170 fig, ax = plt.subplots()
171 ax.quiver(X, Y, u1_sol, u2_sol)

```

```

172 ax.set_xlabel("x")
173 ax.set_ylabel("y")
174 ax.set_title(r"$\vec{u}(x, y)$")
175 plt.savefig("vector_fiel", dpi=500)
176
177 fig, ax = plt.subplots()
178 ax.set_xlabel("x")
179 ax.set_ylabel("y")
180 ax.set_title(r"$\vec{u}(x, y)$")
181 ax.streamplot(X, Y, u1_sol, u2_sol)
182 plt.savefig("stream", dpi=500)
183
184 fig = plt.figure()
185 ax = plt.axes(projection = '3d')
186 ax.set_xlabel("x")
187 ax.set_ylabel("y")
188 ax.set_zlabel(r"$u_1(x, y)$")
189 ax.set_title("Horizontal Velocity")
190 ax.plot_surface(X, Y, u1_sol, cmap="viridis")
191 plt.savefig("horizontal velocity", dpi=500)
192
193 fig = plt.figure()
194 ax = plt.axes(projection = '3d')
195 ax.set_xlabel("x")
196 ax.set_ylabel("y")
197 ax.set_zlabel(r"$u_2(x, y)$")
198 ax.set_title("Vertical Velocity")
199 ax.plot_surface(X, Y, u2_sol, cmap="viridis")
200 plt.savefig("vertical velocity", dpi=500)
201
202 fig = plt.figure()
203 ax = plt.axes(projection = '3d')
204 ax.set_xlabel("x")
205 ax.set_ylabel("y")
206 ax.set_zlabel("p(x, y)")
207 ax.set_title("Pressure")
208 ax.plot_surface(X, Y, p_sol, cmap="viridis")
209 plt.savefig("pressure", dpi=500)
210
211 plt.show()
212
213 #error check
214 error_u1_sol = nrm.l_squ_norm_2d(u1((X, Y)) - u1_sol, (x, y))
215 error_u2_sol = nrm.l_squ_norm_2d(u2((X, Y)) - u2_sol, (x, y))
216 error_p_sol = nrm.l_squ_norm_2d(p((X, Y)) - p_sol, (x, y))
217
218 error_sum = error_u1_sol + error_u2_sol + error_p_sol
219 print(f"{H**2} nodes per function")
220 print(f"u1 error = {error_u1_sol}")
221 print(f"u2 error = {error_u2_sol}")
222 print(f"p error = {error_p_sol}")

```

Listing 5: stokes\_2dim\_ele.py

```

1 """
2 Plotting of 2dim hat function
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 import os
9 import sys
10 current_dir = os.getcwd()
11 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))

```

```

12 sys.path.append(top_dir)
13
14 from tools import basis_functions as bf
15
16 N = 31
17 x = np.linspace(0, 1, N)
18 y = np.linspace(0, 1, N)
19 X, Y = np.meshgrid(x, y)
20
21 fig = plt.figure()
22 ax = plt.axes(projection ='3d')
23
24 ax.plot_surface(X, Y, bf.hat_2d((0.5, 0.5), (X, Y), .25), cmap="viridis")
25 ax.set_xlabel('x')
26 ax.set_ylabel('y')
27 ax.set_zlabel(r'$\phi_{\frac{1}{2}, \frac{1}{2}}(x, y)$')
28 #plt.savefig("hat_basis_2d.png", dpi=500)
29 plt.show()

```

Listing 6: 2d\_hat\_plot.py

```

1 """
2 Plotting of 2dim quadratic basis functions
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 import os
9 import sys
10 current_dir = os.getcwd()
11 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
12 sys.path.append(top_dir)
13
14 from tools import basis_functions as bf
15
16 N = 200
17 x = np.linspace(0, 1, N)
18 y = np.linspace(0, 1, N)
19 X, Y = np.meshgrid(x, y)
20
21 h = 0.1
22
23 pair_1 = (0.5, 0.6)
24 fig = plt.figure()
25 ax = plt.axes(projection ='3d')
26 ax.plot_surface(X, Y, bf.quad_2d(pair_1, (X, Y), h), cmap="viridis")
27 ax.set_xlabel('x')
28 ax.set_ylabel('y')
29 #plt.savefig(f"quad_basis_2d_{pair_1}.png", dpi=500)
30 plt.show()
31
32 pair_2 = (0.5, 0.5)
33 fig = plt.figure()
34 ax = plt.axes(projection ='3d')
35 ax.plot_surface(X, Y, bf.quad_2d(pair_2, (X, Y), h), cmap="viridis")
36 ax.set_xlabel('x')
37 ax.set_ylabel('y')
38 #plt.savefig(f"quad_basis_2d_{pair_2}.png", dpi=500)
39 plt.show()
40
41 pair_3 = (0.6, 0.5)
42 fig = plt.figure()
43 ax = plt.axes(projection ='3d')
44 ax.plot_surface(X, Y, bf.quad_2d(pair_3, (X, Y), h), cmap="viridis")

```

```

45 ax.set_xlabel('x')
46 ax.set_ylabel('y')
47 #plt.savefig(f"quad_basis_2d_{pair_3}.png", dpi=500)
48 plt.show()
49
50 pair_4 = (0.6, 0.6)
51 fig = plt.figure()
52 ax = plt.axes(projection ='3d')
53 ax.plot_surface(X, Y, bf.quad_2d(pair_4, (X, Y), h), cmap="viridis")
54 ax.set_xlabel('x')
55 ax.set_ylabel('y')
56 #plt.savefig(f"quad_basis_2d_{pair_4}.png", dpi=500)
57 plt.show()

```

Listing 7: 2d\_quad\_plot.py

```

1 """
2 Plotting of sum of hat basis functions to show linear interpolation between their peaks
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 import os
9 import sys
10 current_dir = os.getcwd()
11 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
12 sys.path.append(top_dir)
13
14 from tools import basis_functions as bf
15
16
17 N = 1000
18 x = np.linspace(0, 1, N)
19
20 H = 5
21 vertices = np.linspace(0, 1, H)
22 h = vertices[1] - vertices[0]
23
24 func1 = 1 * bf.phi(.25, x, h)
25 func2 = 3 * bf.phi(.5, x, h)
26 func3 = 2 * bf.phi(.75, x, h)
27 func_sum = func1 + func2 + func3
28
29 plt.plot(x, func_sum, label="sum of bases", linestyle="--")
30 plt.plot(x, func1, label=r"$1\phi_{0.25}(x)$", linestyle="--")
31 plt.plot(x, func2, label=r"$3\phi_{0.5}(x)$", linestyle="-.")
32 plt.plot(x, func3, label=r"$2\phi_{0.75}(x)$", linestyle=":")
33 plt.xlabel("x")
34 plt.legend()
35 #plt.savefig("linear_basis_sum")
36 plt.show()

```

Listing 8: plot\_hat\_basis\_sum.py

```

1 """
2 Plotting of hat basis function and its derivative
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 import os
9 import sys
10 current_dir = os.getcwd()

```

```

11 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
12 sys.path.append(top_dir)
13
14 from tools import basis_functions as bf
15
16
17 N = 200
18 x = np.linspace(0, 1, N)
19 h = .3
20
21 plt.plot(x, bf.hat(.5, x, h), label=r"\phi_{0.5}(x)")
22 plt.plot(x, bf.grad_hat(.5, x, h), label=r"\phi_{0.5}'(x)", linestyle="--")
23 plt.legend()
24 plt.title("the hat function and its derivative")
25 plt.xlabel("x")
26 #plt.savefig("linear_basis")
27 plt.show()

```

Listing 9: plot\_hat\_basis.py

```

1 """
2 Plotting of "even" quadratic basis function
3 """
4
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8
9 import os
10 import sys
11 current_dir = os.getcwd()
12 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
13 sys.path.append(top_dir)
14
15 from tools import basis_functions as bf
16
17 N = 1000
18 x = np.linspace(0, 1, N)
19
20 H = 6
21 vertices = np.linspace(0, 1, H)
22 h = vertices[1] - vertices[0]
23
24 func1 = 1 * bf.quad_e(.5, x, h)
25
26 plt.plot(x, func1)
27 plt.xlabel("x")
28 #plt.savefig("quad_basis_even")
29 plt.show()

```

Listing 10: plot\_quad\_basis\_even.py

```

1 """
2 Plotting of "odd" quadratic basis function
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 import os
9 import sys
10 current_dir = os.getcwd()
11 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
12 sys.path.append(top_dir)
13

```

```

14 from tools import basis_functions as bf
15
16 N = 1000
17 x = np.linspace(0, 1, N)
18
19 H = 6
20 vertices = np.linspace(0, 1, H)
21 h = vertices[1] - vertices[0]
22
23 func2 = 1 * bf.quad_o(.5, x, h)
24
25 plt.plot(x, func2)
26 plt.xlabel("x")
27 #plt.savefig("quad_basis_odd")
28 plt.show()

```

Listing 11: plot\_quad\_basis\_odd.py

```

1 """
2 Plotting of a sum of quadratic basis functions scaled by some numbers to show
3     quadratic interpolation for the element
4 """
5
6 import numpy as np
7 import matplotlib.pyplot as plt
8
9 import os
10 import sys
11 current_dir = os.getcwd()
12 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
13 sys.path.append(top_dir)
14
15 from tools import basis_functions as bf
16
17 N = 1000
18 x = np.linspace(0, 1, N)
19
20 H = 3
21 vertices = np.linspace(0, 1, H)
22 h = vertices[1] - vertices[0]
23
24 func1 = 1 * bf.quad(0, x, h)
25 func2 = 3 * bf.quad(.5, x, h)
26 func3 = 3 * bf.quad(1, x, h)
27 func_sum = func1 + func2 + func3
28
29 plt.plot(x, func1, label=r"$ 1 \backslash \psi_0(x) $", linestyle="--")
30 plt.plot(x, func2, label=r"$ 3 \backslash \psi_{0.5}(x) $", linestyle="--")
31 plt.plot(x, func3, label=r"$ 3 \backslash \psi_1(x) $", linestyle="--")
32 plt.plot(x, func_sum, label="sum of bases")
33
34 plt.xlabel("x")
35 plt.legend()
36 #plt.savefig("quad_basis")
37 plt.show()

```

Listing 12: plot\_quad\_basis\_funky.py

```

1 """
2 Plotting of a sum of quadratic basis functions on one element
3 """
4
5 import numpy as np
6 import matplotlib.pyplot as plt
7

```

```

8 import os
9 import sys
10 current_dir = os.getcwd()
11 top_dir = os.path.abspath(os.path.join(current_dir, "..", ".."))
12 sys.path.append(top_dir)
13
14 from tools import basis_functions as bf
15
16 N = 1000
17 x = np.linspace(0, 1, N)
18
19 H = 3
20 vertices = np.linspace(0, 1, H)
21 h = vertices[1] - vertices[0]
22
23 func1 = 1 * bf.quad(0, x, h)
24 func2 = 1 * bf.quad(.5, x, h)
25 func3 = 1 * bf.quad(1, x, h)
26 func_sum = func1 + func2 + func3
27
28 plt.plot(x, func1, label=r"\psi_0(x)", linestyle="--")
29 plt.plot(x, func2, label=r"\psi_{0.5}(x)", linestyle="--")
30 plt.plot(x, func3, label=r"\psi_1(x)", linestyle="--")
31 plt.plot(x, func_sum, label="sum of bases")
32
33 plt.xlabel("x")
34 plt.legend()
35 #plt.savefig("quad_basis")
36 plt.show()

```

Listing 13: plot\_quad\_basis\_sum.py

```

1 import numpy as np
2
3 """----- LINEAR BASIS FUNCTIONS -----"""
4
5 epsilon = 1e-3
6
7 def hat(i, x, h):
8     """hat function with peak at i.
9
10    Args:
11        i (float): position of peak
12        x (float): variable
13
14    Returns:
15        float:
16    """
17    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])
18
19
20 def grad_hat(i, x, h):
21     """derivative of hat function with peak at i
22
23    Args:
24        i (float): position of peak
25        x (float): variable
26
27    Returns:
28        float:
29    """
30    return np.piecewise(x, [x <= (i-h), (x > (i-h))&(x <= i), (x > (i))&(x < (i+h)), x >=(i+h)], [0 , lambda x: 1/h , lambda x: -1/h , 0])
31

```

```

32     >=(i+h)], [0 , 1/h , -1/h
33     , 0 ])
34
35 def hat_2d(i, x, h):
36     return hat(i[0], x[0], h) * hat(i[1], x[1], h)
37
38 def phi_and_hat_2d(i, x, h, hat_bdry=[]):
39     return phi(i[0], x[0], h) * phi(i[1], x[1], h, hat_bdry)
40
41
42 def phi(i, x, h, boundary=[0, 1]):
43     """Piecewise linear basis function
44
45     Args:
46         i (float): position of peak
47         x (float): variable
48         boundary (tuple): boundary conditions on
49
50     Returns:
51         float:
52     """
53
54     if (i == np.array(boundary)).any(): #np.abs(i - 0) < epsilon or np.abs(i - 1) <
55         epsilon:
56             return 0*x
57     else:
58         return hat(i, x, h)
59
60 def grad_phi(i, x, h, boundary=[0, 1]):
61     """derivative of hat function with peak at i
62
63     Args:
64         i (float): position of peak
65         x (float): variable
66
67     Returns:
68         float:
69     """
70     if (i == np.array(boundary)).any(): #np.abs(i - 0) < epsilon or np.abs(i - 1) <
71         epsilon:
72             return 0*x
73     else:
74         return np.piecewise(x, [x <= (i-h), (x > (i-h))&(x <= i), (x > (i))&(x < (i+h))
75 ), x>=(i+h)], [0 , 1/h , -1/h
76     , 0 ])
77
78 def phi_2d(i, x, h):
79     """2 dimensional hat function with peak at i.
80
81     Args:
82         i (tuple): position of peak
83         x (tuple): variable
84
85     Returns:
86         float:
87     """
88
89     return phi(i[0], x[0], h) * phi(i[1], x[1], h)
90
91 # some derivatives of previously declared functions
92 def dx_phi_2d(i, x, h):
93     return grad_phi(i[0], x[0], h) * phi(i[1], x[1], h)

```

```

91 def dy_phi_2d(i, x, h):
92     return phi(i[0], x[0], h) * grad_phi(i[1], x[1], h)
93
94 def dx_hat_2d(i, x, h):
95     return grad_hat(i[0], x[0], h) * hat(i[1], x[1], h)
96
97 def dy_hat_2d(i, x, h):
98     return hat(i[0], x[0], h) * grad_hat(i[1], x[1], h)
99
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
119 def quad_e(i, x, h):
120     """function to generate odd components of quadratic basis
121
122     Args:
123         i (_float_): position of peak
124         x (_float_): variable
125         h (_type_): width
126
127     Returns:
128         float:
129     """
130     return np.piecewise(x, [x <= (i-(2*h)), (x > (i-(2*h))) & (x <= i)
131                         , (x > i) & (x < (i+(2*h))) , x>=(i+(2*h))],
132                         [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
133 ])
134
135 def quad(i, x, h):
136     """function for vertices of quadratic basis elements
137
138     Args:
139         i (_float_): position of peak
140         x (_float_): variable
141         h (_type_): width
142
143     Returns:
144         float:
145     """
146
147     if (round(i/h) % 2) == 1:
148         return quad_o(i, x, h)
149     else:
150         return quad_e(i, x, h)
151
152 def quad_2d(i, x, h):
153     """2 dimensional quadratic basis function

```

```

152     Args:
153         i (_float_): position of peak
154         x (_float_): variable
155         h (_type_): width
156
157     Returns:
158         float:
159     """
160     return quad(i[0], x[0], h) * quad(i[1], x[1], h)
161
162
163 def psi(i, x, h, boundary=[0, 1]):
164     """Piecewise Quadratic basis function with peak at i and element of size 2*h
165
166     Args:
167         i (float): peak of function
168         x (float): variable
169         h (float): half the width of element
170         boundary (list, optional): boundary of domain. Defaults to [0, 1].
171
172     Returns:
173         float:
174     """
175     if (i == np.array(boundary)).any():
176         return 0*x
177     else:
178         return quad(i, x, h)
179
180 def grad_psi(i, x, h, boundary=[0, 1]):
181     """Piecewise quadratic basis function with peak at i and element of size 2*h
182
183     Args:
184         i (float): peak of function
185         x (float): variable
186         h (float): half the width of element
187         boundary (list, optional): boundary of domain. Defaults to [0, 1].
188
189     Returns:
190         float:
191     """
192     if (i == np.array(boundary)).any():
193         return 0*x
194     else:
195         return np.gradient(quad(i, x, h), h)
196
197
198 def psi_2d(i, x, h):
199     """Piecewise Quadratic 2 dimensional basis function with peak at i and element of
size (2*h)**2
200
201     Args:
202         i (float): peak of function
203         x (float): variable
204         h (float): half the width of element
205
206     Returns:
207         float:
208     """
209     return psi(i[0], x[0], h) * psi(i[1], x[1], h)
210
211 def dx_psi_2d(i, x, h):
212     """the derivative with respect to x of a 2d quadratic basis function
213
214     Args:
215         i (float): peak of function

```

```

216     x (float): variable
217     h (float): half the width of element
218
219     Returns:
220         float:
221     """
222
223     return grad_psi(i[0], x[0], h) * psi(i[1], x[1], h)
224
225 def dy_psi_2d(i, x, h):
226     """the derivative with respect to y of a 2d quadratic basis function
227
228     Args:
229         i (float): peak of function
230         x (float): variable
231         h (float): half the width of element
232
233     Returns:
234         float:
235     """
236
237     return psi(i[0], x[0], h) * grad_psi(i[1], x[1], h)
238
239
240 """----- EXTRA FUNCTIONS -----"""
241
242
243 def conv_sol(solution, x, basis_func, vertices, h):
244     """Converts a solution vector (an array) into an interpolated function on x with
245     corresponding
246     basis functions (basis_func).
247
248     Args:
249         solution (numpy array): solution vector u to matrix equation Au=F
250         x (numpy array): domain over which we want this new function over
251         basis_func (function): one of the basis functions. The function is a variable
252         here so do not
253             put in any variables into it.
254         vertices (list): list of vertices which correspond to each entry of the
255             solution vector u
256             h (float): width of element. This is the same h as you would have used in
257             defining your mesh
258
259     Returns:
260         numpy array:
261     """
262
263     u = 0
264     for i, vert in enumerate(vertices):
265         u = u + (solution[i] * basis_func(vert, x, h))
266
267     return u

```

Listing 14: basis\_functions.py

```

1 """
2 Defining norms on functions to find error in calculations
3 """
4
5 import numpy as np
6
7 def l_squ_norm(f, x0):
8     """The l squared norm for functions of one variable, or here for 1 dimensional
9     arrays.
10
11     Args:

```

```

11     f (numpy array): function given by numpy array
12     x0 (numpy array): array on which the function is defined on
13
14     Returns:
15         float:
16     """
17     return np.trapz( np.abs(f)**2 , x0, axis=0)
18
19 def l_squ_norm_2d(f, x0):
20     """The l squared norm for functions of two variable, or here for 2 dimensional
21     arrays.
22
23     Args:
24         f (numpy array): function given by numpy array
25         x0 (numpy array): tuple of arrays on which the function is defined on
26
27     Returns:
28         float:
29     """
30
31     return np.sqrt( np.trapz(np.trapz( np.abs(f)**2 , x0[0], axis=0), x0[1], axis=0) )

```

Listing 15: norms.py

```

1 import numpy as np
2 #import basis_functions as bf
3
4 import os
5 import sys
6 current_dir = os.getcwd()
7 top_dir = os.path.abspath(os.path.join(current_dir, ".."))
8 sys.path.append(top_dir)
9
10 from tools import basis_functions as bf
11
12 """----- APPROXIMATE VECTOR PRODUCTS
13 -----"""
14 # this section contains functions which will predominantly use the numpy gradient
15 # function to calculate derivatives
16
17 def grad_dot_grad(func0, func1, h):
18     """returns a dot product of gradients of two functions of two variables.
19     ie, grad(func0) dot grad(func1)
20
21     Args:
22         func0 (function): a numpy function of two variables
23         func1 (function): a numpy function of two variables
24
25     Returns:
26         numpy array:
27     """
28
29     dy_f0, dx_f0 = np.gradient(func0, h)
30     dy_f1, dx_f1 = np.gradient(func1, h)
31
32     return np.multiply(dx_f0, dx_f1) + np.multiply(dy_f0, dy_f1)
33
34 def func_mul_pdfunc(func0, func1, h, axis):
35     """returns a product of a function and a partial derivative of another function.
36     ie, func0 * partial_derivative(func1)
37
38     Args:
39         func0 (function): a numpy function of two variables
40         func1 (function): a numpy function of two variables
41         axis (0 or 1): pick 1 for partial derivative with respect to x or 0 for

```

```

    partial derivative with respect to y
40
41     Returns:
42         numpy array:
43     """
44     pd_f1 = np.gradient(func1, h, axis=axis)
45
46     return np.multiply(pd_f1, func0)
47
48
49 """----- PRECISE VECTOR PRODUCTS
50 -----"""
51 # this section contains precise evaluations of important vector products using pen and
52 # paper maths as much as possible
53
54 def grad_dot_grad_phi2d(vert0, vert1, X, h):
55     """returns a dot product of gradients of two bf.phi_2d functions.
56     mathematically --> grad(phi2d) dot grad(phi2d)
57
58     Args:
59         vert0 (float): peak of associated basis function
60         vert1 (float): peak of associated basis function
61         X (numpy array): array to evaluate functions on
62         h (float): width between nodes
63
64     Returns:
65         numpy array:
66     """
67
68     x, y = X
69     i, j = vert0
70     a, b = vert1
71
72     dx_0 = bf.grad_phi(i, x, h) * bf.phi(j, y, h)
73     dy_0 = bf.phi(i, x, h)      * bf.grad_phi(j, y, h)
74
75     dx_1 = bf.grad_phi(a, x, h) * bf.phi(b, y, h)
76     dy_1 = bf.phi(a, x, h)      * bf.grad_phi(b, y, h)
77
78     return dx_0*dx_1 + dy_0*dy_1
79
80 def grad_dot_grad_psi2d(vert0, vert1, X, h):
81     """returns a dot product of gradients of two bf.psi_2d functions.
82     mathematically --> grad(psi2d) dot grad(psi2d)
83
84     Args:
85         vert0 (float): peak of associated basis function
86         vert1 (float): peak of associated basis function
87         X (numpy array): array to evaluate functions on
88         h (float): width between nodes
89
90     Returns:
91         numpy array:
92     """
93
94     x, y = X
95     i, j = vert0
96     a, b = vert1
97
98     dx_0 = bf.grad_psi(i, x, h) * bf.psi(j, y, h)
99     dy_0 = bf.psi(i, x, h)      * bf.grad_psi(j, y, h)
100
101    dx_1 = bf.grad_psi(a, x, h) * bf.psi(b, y, h)
102    dy_1 = bf.psi(a, x, h)      * bf.grad_psi(b, y, h)

```

```

102     return dx_0*dx_1 + dy_0*dy_1
103
104 def grad_dot_grad_hat2d(vert0, vert1, X, h):
105     """returns a dot product of gradients of two bf.hat_2d functions.
106     mathematically --> grad(psi2d) dot grad(psi2d)
107
108     Args:
109         vert0 (float): peak of associated basis function
110         vert1 (float): peak of associated basis function
111         X (numpy array): array to evaluate functions on
112         h (float): width between nodes
113
114     Returns:
115         numpy array:
116     """
117
118     x, y = X
119     i, j = vert0
120     a, b = vert1
121
122     dx_0 = bf.grad_hat(i, x, h) * bf.hat(j, y, h)
123     dy_0 = bf.hat(i, x, h) * bf.grad_hat(j, y, h)
124
125     dx_1 = bf.grad_hat(a, x, h) * bf.hat(b, y, h)
126     dy_1 = bf.hat(a, x, h) * bf.grad_hat(b, y, h)
127
128     return (dx_0*dx_1) + (dy_0*dy_1)
129
130 def gdg_phi_and_hat_2d(vert0, vert1, X, h, hat_bdry=[]):
131     """returns a dot product of gradients of two bf.hat_2d functions.
132     mathematically --> grad(phi2d) dot grad(psi2d)
133
134     Args:
135         vert0 (float): peak of associated basis function
136         vert1 (float): peak of associated basis function
137         X (numpy array): array to evaluate functions on
138         h (float): width between nodes
139         hat_bdry (list): location of boundary for hat functions
140
141     Returns:
142         numpy array:
143     """
144
145     x, y = X
146     i, j = vert0
147     a, b = vert1
148
149     dx_0 = bf.grad_phi(i, x, h) * bf.phi(j, y, h, hat_bdry)
150     dy_0 = bf.phi(i, x, h) * bf.grad_phi(j, y, h, hat_bdry)
151
152     dx_1 = bf.grad_phi(a, x, h) * bf.phi(b, y, h, hat_bdry)
153     dy_1 = bf.phi(a, x, h) * bf.grad_phi(b, y, h, hat_bdry)
154
155     return (dx_0*dx_1) + (dy_0*dy_1)

```

Listing 16: vector\_products.py