Mesh partial differential equations (PDEs): application and implementation

Róisín Hill, Niall Madden,
School of Mathematics, Statistics and Applied Mathematics,
National University of Ireland, Galway, Ireland.

July 24, 2020

Abstract

Niall: Eventually, this abstract needs to be more direct and detailed... Also, needs to highlight Algorithm 2, to emphasise the novelty

We present a novel method for generating layer-adapted meshes using a combination of h- and r-refinement, specifically using mesh PDEs (MPDEs). We show how MPDEs can be used to create Bakhvalov type meshes that are appropriate to use when solving one-dimensional singularly-perturbed problems whose solutions exhibit layers and sublayers. We demonstrate how to extend this method to generate two-dimensional, non-tensor product, layer-adapted meshes. These meshes are a suitable tool to use when solving PDEs where diffusion varies spatially. We present algorithms and Python code to implement these methods in FEniCS [1]. Numerical examples with parameter robust solutions are included to support our methods.

§1. Introduction; Keep "generic" stuff, and move that which is specific to 1D to Section 2(?). I expect, we'll combine FEM and FEniCS.

§2. 1D
- 2.1 Model problem (with boundary layers)
- 2.2 FEM
- 2.3 Necessity of layer-adapted meshes (with B-mesh?)
- 2.4 MPDE+B-mesh
- 2.5 Algorithms
- 2.6 Examples

§3. 2D

Temp: outline of paper

1 Introduction

[1] [2] Our goal is to present a method, to enable the reader to efficiently generate qualitatively and quantitatively accurate numerical solutions to differential equations. Often solutions to differential equations have features of interest, for example, regions where there is a rapid change in the solution. We use mesh PDEs (MPDEs) to generate non-uniform meshes that are suitably concentrated so as to resolve these features. We will focus on differential equations whose solutions exhibit layers, primarily because they are an important class of problem for which highly non-uniform meshes are used. We present algorithms to generate these meshes and Python code to implement them in FEniCS [13]. In addition, we present numerical results, demonstrating how these algorithms can be applied to both one-dimensional and two-dimensional problems. We discuss differential equations in Section 1.1, in particular, singularly perturbed differential equations. In Section 2 we present our model one-dimensional problems. We show how to obtain solutions to these problems using finite element methods (FEMs), how to implement our method in FEniCS and outline the software system in Section 2. Then, in Section 2.2, we show why layer resolving meshes are important, presenting a particularly important example, the graded *Bakhvalov mesh* [17]. MPDEs are introduced in Section 2.3 we show why they are appropriate as the basis for algorithms to drive mesh adaption. In Section 2.3 we describe how MPDEs can be used to generate one-dimensional Bakhvalov type meshes. We present two different algorithms for generating these meshes in Section 2.4. Numerical results are given in Section 2.5 for singularly-perturbed reaction-diffusion equations, including a coupled system of equations. Our Python code for solving a differential equation on a uniform mesh in FEniCS is included in Appendix A. The code for generating one-dimensional Bakhvalov meshes using an MPDE is in Appendix B. In Section 3 we describe how the

[1] Niall: Need to rewrite all this later, but only after the rest of the paper is finished
[2] Niall: Keep an eye on when abbreviations, such as ODE, PDE and FEM are introduced and used. RH: done.

method can be extended to generate two-dimensional layer-adapted meshes. We focus on a two-dimensional reaction-diffusion equation with diffusion that varies spatially. We present an algorithm for generating these meshes in Section 3.2, and numerical results in Section 3.3. Python code for generating two-dimensional layer-adapted meshes using an MPDE is in Appendix C.[3] We use the notation presented in [4][§2.1] and it's Glossary of symbols: Notation for specific vector spaces.

1.1 Differential equations

Differential equations arise in many aspects of science and engineering. They model how phenomena change over time or space, such as how a cancerous tumour grows, how a string vibrates, or how a pollutant disperses. So their solution is of fundamental importance to science and engineering. Analytical solutions to differential equations are rarely available so we want to calculate reliable approximate solutions to these problems using numerical schemes. Often there is a subsection of the solution that is of particular interest and we want to resolve these small scales effects without excessive computational cost.

Singularly perturbed problems are differential equations whose leading term is multiplied by a small positive constant, usually denoted ε. They are singularly perturbed in the sense that they are well posed for all non-zero values of ε, but ill-posed if one formally sets $\varepsilon = 0$. For a more formal definition see, e.g., [17, p.2] or [11, p.2-3]. Solutions to singularly perturbed problems typically exhibit boundary layers and may also contain interior layers. The elliptic singularly perturbed differential reaction-diffusion equations that we focus on in this paper are

$$-\varepsilon^2 \Delta u + ru = f \quad \text{in } \Omega \subseteq \mathbb{R}^d, \quad \text{with } u|_{\partial\Omega} = 0, \tag{1}$$

where $d = 1, 2$. Here r and f are given functions, $\varepsilon > 0$ and $r \geq \beta^2$ on $\overline{\Omega}$, where β is some positive constant.

1.2 Numerical methods and mesh adaptivity

[4] To ensure the layer regions of singularly-perturbed problems are resolved when solving them numerically, the local mesh width needs to be very small, in a way that depends on ε. Often $\varepsilon \ll 1$ and consequentially, uniform meshes are unsuitable.

Since, uniform meshes are usually not appropriate, we use adapted meshes. There are three primary strategies for generating adapted meshes: h-refinement, p-refinement and r-refinement. The most extensively used are the h-refinement methods which adapt an initial mesh by adding mesh points to create a finer mesh in the layer regions. In p-refinement methods the order of the elements is increased in areas of interest without changing the number of mesh points. However, r-refinement methods, where our interest lies, relocates mesh points to control the error in the solution while fixing the mesh topology and the number of mesh points. Our objective is to compute "parameter-robust" solutions, on meshes generated using an r-refinement method. That is, solutions that have errors of similar magnitude and order of convergence independent of the value of ε.

[3] Niall: Suggest reading through some books, and picking a good introductory one, and then follow its notation. SIAM, if possible? RH: Is this sufficient -Section 1

[4] NM: Ignore for now, and maybe rewrite later

2 One-dimensional problems

2.1 A scalar problem, and its FEM solution

Our first boundary layer problem is the following one-dimensional singularly perturbed reaction-diffusion equation

$$-\varepsilon^2 u''(x) + ru(x) = f, \quad \text{for } x \in \Omega := (a,b), \tag{2a}$$

with boundary conditions,

$$u(a) = u(b) = 0, \tag{2b}$$

where $0 < \varepsilon \ll 1$, r and f are given functions and $r \geq \beta^2 > 0$ where β is some positive constant. In spite of its simplicity, this is a very frequently studied problem. With reasonable assumptions on f it features boundary layers at one or both boundaries. Indeed, it is one of the problems solved by the first boundary-layer adapted mesh of [3]. That paper applied a finite difference scheme, but here we use FEMs, which we now describe, along the a simple FEniCS implementation.

The bilinear form, associated with (2) is

$$a(u,v) := \varepsilon^2 (u',v') + (ru,v), \tag{3}$$

[5] where $(\cdot,\cdot)$ is the usual L^2 inner product. Then the weak form of (2) is: *find* $u \in H_0^1(\Omega)$, *such that*[6]

$$a(u,v) = (f,v), \quad \text{for all } v \in H_0^1(\Omega). \tag{4}$$

We obtain a finite element method by restricting the choice of u and v to a finite dimensional subspace, V_h, of $H_0^1(\Omega)$. That is, we find $u_h \in V_h$, such that

$$a(u_h,v_h) = (f,v_h), \quad \text{for all } v_h \in V_h. \tag{5}$$

Specifically, we compute the coefficients of the u_h, with respect to some basis, that solves (5).

To be more precise, we will focus on the simplest finite element method: the Galerkin method with piecewise linear basis functions ($\mathcal{P}_1$-FEM). On an arbitrary mesh $a = x_0 < x_1 < \cdots < x_N = b$ define the usual "hat functions" $\{\psi_i\}_{i=1}^{N-1}$ as

$$\psi_i(x) = \begin{cases} (x - x_{i-1})/(x_i - x_{i-1}) & \text{if } x_{i-1} \leq x < x_i, \\ (x_{i+1} - x)/(x_{i+1} - x_i) & \text{if } x_i \leq x \leq x_{i+1}, \\ 0 & \quad\text{otherwise.} \end{cases}$$

V_h is the space spanned by these. Then, u_h can be written as

$$u_h = \sum_{i=1}^{N-1} \lambda_i \psi_i.$$

[7] Taking

$$v_h = \psi_i \quad \text{for } i = 1, \ldots, N-1,$$

in (5) yields $N-1$ equations, which can be solved to give the λ_i.

[5]RH: If I use $a(\cdot,\cdot)$ to denote the bilinear form is it okay to also use the interval (a,b)?

[6]Niall: Check $H_0^1(\Omega)$ and other places domain is specified. RH: done.

[7]Do we need to do something similar in 2D. Niall: Suggest find a good standard reference, for both 1D and 2D

The Galerkin FEM is very powerful, and yet can be quite complicated to implement. It is powerful in the sense that it generalises to many classes of PDEs, domains of various dimensions and shapes, and incorporates many spaces of approximating functions, all within the same theoretical framework. Due to the "power of abstraction" [6] the mathematical formulation of the method is essentially the same regardless of these possible complications.

However, FEMs can be quite complicated to implement. One must discretise the domain and compute transformations of quantities onto a reference element ([7]); almost always quadrature is required to compute the associated integrals. Contributions from the elements are assembled to generate a system of linearly independent algebraic equations [18]. Finally, this system of equations is solved; careful selection of suitable solvers for larger problems usually necessary. Normally, code to visualise the solution and calculate *a posteriori* error estimates is also needed.

The code needed differs depending on the order of the basis functions, the shape and dimension of the domain and it's elements. The power of abstraction seen in the mathematical notation does not apply to the code required to find the numerical solution, unless we uses purpose built software. which automate many of the tasks needed to implement FEMs. Notable examples include FEniCS [1], deal.II [2], MFEM [14], FreeFEM [8], and Firedrake [15].

Our focus is on the use of FEniCS, a free, open-source software system, that automates many aspects of implementing FEMs while still allowing the user access to lower-level features. Using either the Python or C++ interfaces, the user specifies the domain and how to discretise it (i.e., define a mesh), and their choice of basis functions to use. Then, they define the weak form of the problem using a syntax that is very similar to the mathematical formulation. Other components of the suite then take care of assembling and solving the related system of algebraic equations.

To demonstrate a simple use of FEniCS we consider the following specific example of (2):

$$-\varepsilon^2 u''(x) + u(x) = 1 - x, \text{ for } x \in \Omega := (0,1), \qquad \text{with } u(0) = u(1) = 0, \tag{6}$$

where $0 < \varepsilon \ll 1$; its solution exhibits a layer near $x = 0$.

In Listing 1, we show a snippet of FEniCS code, the syntax is similar to the mathematical formulation of (4).

Listing 1: The weak form of (6) for FEniCS.

```
f = Expression ('1-x[0]', degree = 2)
a = epsilon*epsilon*inner(grad(u), grad(v))*dx + inner(u,v)*dx
L = f*v*dx
solve(a==L, uN, bc) # Solve, applying the boundary conditions strongly
```

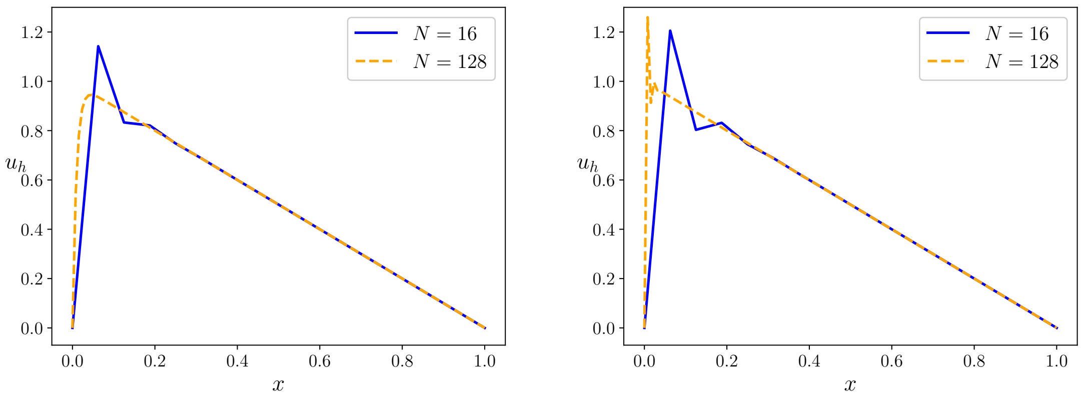
In Appendix A[8] we present the complete Python code for solving the one-dimensional singularly perturbed reaction-diffusion equation (6) in FEniCS on a uniform mesh.

2.2 The necessity of layer resolving meshes

A useful numerical solution to (6) should be both qualitatively and quantitatively representative of the exact solution. That is, it should accurately determine both the layer location and width. However, consider the numerical solutions to (6), shown in Figure 1, which were generated using a uniform mesh. One can see that the solutions change rapidly near $x = 0$, but outside this layer region it closely resembles the solution

[8]Niall: Review code and comments; include author, and file name; RH: done. time to plan how/where to publish the code

to (6) with $\varepsilon = 0$ and the boundary conditions neglected, viz, $1 - x$. In Figure 1(a), $\varepsilon = 10^{-2}$, and one can see that a mesh with 16 intervals is not adequate to resolve the layer region, but on a mesh with $N = 128$ intervals the layer is resolved, but only because N is $\mathcal{O}(\varepsilon^{-1})$. This is clear from Figure 1(b) where one can observe that, when $\varepsilon = 10^{-4}$, the solution with $N = 128$ is unstable, and fails to resolve the layer. In general, a satisfactory solution is obtained with a uniform mesh only when N is $\mathcal{O}(\varepsilon^{-1})$, which is infeasible since we wish to accurately solve this problem for any positive ε, irrespective of how small it is. [9]



(a) $\varepsilon = 10^{-2}$. (b) $\varepsilon = 10^{-4}$.

Figure 1: FEM solutions to (6) on uniform meshes.

Obviously, for a problem such as (6) we need a mesh that is very fine in the layer region. One strategy for generating such a mesh is to use asymptotic information concerning the solution (and its derivatives) and, specifically the location and width of the layer. Where this information is available, one can construct *a priori* fitted layer-resolving meshes. Among the class of such meshes that have been successfully applied to solving singularly perturbed problems, the piecewise uniform mesh of Shishkin [19] and the graded mesh of Bakhvalov [3] are the most widely studied. Both these methods yield parameter-robust solutions.[10]

Recall that a mesh is a partition of the domain into intervals in one-dimension, and triangles or quadrilaterals in two-dimensions. We denote a generic one-dimensional mesh, with N intervals on $[a, b]$, as $\omega : a = x_0 < x_1 < \cdots < x_{N-1} < x_N = b$. One can describe a mesh by explicitly listing the coordinates of each x_i, Here, we will describe a mesh in terms of a "mesh generating function".

Definition 2.1 (Mesh generating function)**.** A mesh generating function is a strictly monotonic bijective function $\varphi : \overline{\Omega}^{[c]} := [0, 1] \to \overline{\Omega} := [a, b]$ that maps a uniform mesh $\xi_i = i/N$, for $i = 0, 1, \dots, N$, to a potentially non-uniform mesh $x_i = \varphi(i/N)$, for $i = 0, 1, \dots, N$, with $\varphi(0) = a$ and $\varphi(1) = b$.

For example, if $\varphi(\xi) = \xi$, then the resulting mesh will be a uniform mesh on the interval $\overline{\Omega}$ as shown in Figure 2. If $\varphi(\xi) = \xi^4$, then the resulting mesh is graded on the interval $\overline{\Omega}$, with mesh points closest together when ξ is close to 0, as shown in Figure 3.

[9]Niall: Humour me, and add, just for comparison, a version of Figure 1(a) but with $\varepsilon = 10^{-4}$. RH: Done, I think I prefer it.

[10]Niall: Have we explained what parameter-robust means? RH: Yes, see last line of Section 1.2

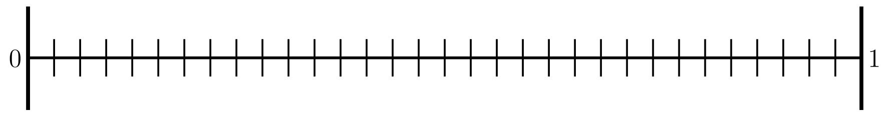


Figure 2: Mesh generated when $\varphi(\xi) = \xi$.

[11]

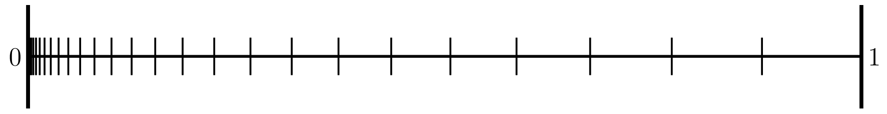


Figure 3: Mesh generated when $\varphi(\xi) = \xi^4$.

The first layer adapted mesh for singularly perturbed problems was proposed in [3]. It is graded, and very fine, in layer regions, and uniform elsewhere. We will now outline its construction based on the presentation in [11]. To motivate the mesh we will consider the reaction-diffusion problem (6), whose solution has a single boundary layer which is located near $x = 0$. Let $\exp(-\beta x/\varepsilon)$ be a function that represents this boundary layer. Then the mesh points x_i near $x = 0$ are chosen to satisfy

$$\exp\left(\frac{-\beta x_i}{\sigma\varepsilon}\right) = 1 - \frac{i}{qN} \quad \text{for } i = 0, 1, \dots, \tag{7}$$

where $q \in (0, 1)$ is roughly the portion of mesh points used to resolve the layer, $\sigma > 0$ is a constant that depends on the underlying method, typically chosen to be the formal order of the underlying method, and N is the number of mesh intervals.

The mesh is expressed in terms of the mesh generating function,

$$\varphi(t) = \begin{cases} \chi(t) := -\dfrac{\sigma\varepsilon}{\beta}\ln\left(1 - \dfrac{t}{q}\right), & \text{for } t \in [0, \tau], \\ \pi(t) := \chi(\tau) + (t - \tau)\varphi'(\tau), & \text{elsewhere.} \end{cases} \tag{8}$$

The mesh generated has mesh points $x_i = \varphi(i/N)$ for $i = 0, 1, \dots, N$.

We measure the error in our numerical solution in the energy norm,

$$\|u\|_{\mathrm{E},\Omega} = \sqrt{a(u, u)},$$

with

$$e_h(x) = |u_{2h}(x) - u_h(x)|,$$

where u_h is the $\mathcal{P}_1$-FEM solution and u_{2h} is the $\mathcal{P}_2$-FEM solution on the same mesh, that is, the Galerkin method with quadratic basis functions. A proof that the error in the $\mathcal{P}_1$-FEM solutions of singularly-perturbed problems, calculated on Bakhvalov meshes, measured in the energy norm converges linearly is given in [16].

[12]

[11]RH: Does this image add anything?

[12]Niall: Need to think if error estimates are appropriate here? What is a good source? RH: Still not quite right.!!!

2.3 MPDEs

We are interested in developing parameter-robust methods, where the quality of the solution is independent of the value of the perturbation parameter, ε. One way to achieve this is to use layer adaptive methods which generate meshes that concentrate mesh points in regions where large variations in the solution occur. There are various ways one can express a formula for such a non-uniform mesh. In this article we define meshes in terms of a *"mesh generating function"*.

A mesh PDE, is presented in [9], as a way of performing r-refinement. First, a PDE whose solution is a mesh generating function is posed. The PDE features a coefficient that controls the concentration of points in the resulting mesh. As we will see it may be defined explicitly in a way that generates *a priori* layer adaptive meshes, or in terms of error estimates.

If one wants to find the solution to (2) on a fixed number of mesh points, how should these points be selected? One approach is to choose *a mesh density function*, $\rho : \overline{\Omega} \to \mathbb{R}_{>0}$ and then construct a mesh so that the integral of ρ is the same on each mesh interval. This is known as the *"equidistribution principle"* and was introduced by [5]. A mesh $\omega : a = x_0 < x_1 < \cdots < x_N = b$, with N intervals, equidistributes ρ when

$$\int_{x_i}^{x_{i+1}} \rho(x)dx = \frac{1}{N}\int_a^b \rho(x)dx, \quad \text{for all} \quad i = 0, 1, \ldots, N. \tag{9a}$$

or, equivalently,

$$\int_a^{x_i} \rho(x)dx = \frac{i}{N}\int_a^b \rho(x)dx \quad \text{for } i = 0, 1, \ldots, N, \tag{9b}$$

If we choose $\rho = C$, a constant, then $x(\xi) = \xi$ is a solution to (9a) and the mesh generated is uniform, see Figure 2. More typically, ρ is not constant and the resulting mesh is finer where ρ is large. For example, if $\rho = 1/\xi^3$, then $x(\xi) = \xi^4$ is a solution to (9a) and the mesh generated is as shown in Figure 3.

Since the equidistribution problem (9b) is nonlinear, typically one must solve it using an iterative process until the resulting mesh equidistributes ρ. One way to think of layer-adapted meshes is that if the numerical solution, u_h, is calculated on the adapted mesh, then the mesh will be finer in the layer region than elsewhere on the domain. We solve the MPDE on a computational domain $\Omega^{[c]}$ using a uniform mesh and the PDE on a physical domain Ω using the layer-adapted mesh generated from the solution to the MDPE. If $u_h(x)$ is transformed from Ω onto $\Omega^{[c]}$, the layer region will be stretched and no sharp layer exists in $\bar{u}_h(x(\xi))$, as can be seen in Figure 4.

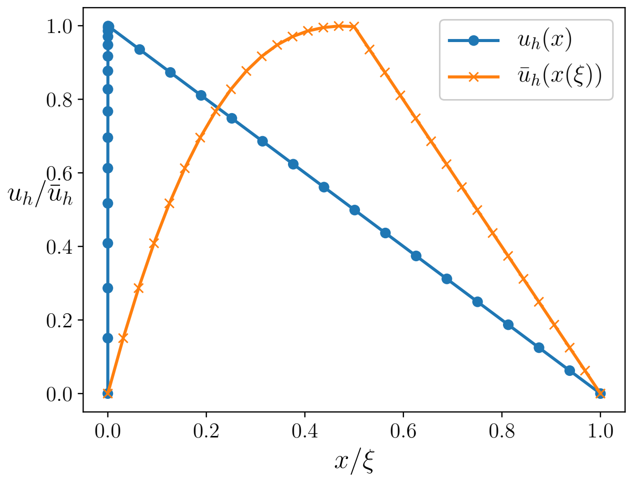


Figure 4: Solution to (6) with $\varepsilon = 10^{-4}$ and $N = 32$ generated on an adapted mesh $\{x\}$, and the solution transformed onto a uniform mesh $\{\xi\}$.

We can derive an MPDE by considering the equidistribution principle as a mapping $x(\xi) : \overline{\Omega}^{[c]} := [0,1] \to \overline{\Omega} := [a,b]$ from the computational coordinate ξ to a physical coordinate x, which satisfies

$$\int_a^{x(\xi)} \rho(x)dx = \xi \int_a^b \rho(x)dx. \tag{10}$$

First, we differentiate (10) with respect to ξ. We can see that $x(\xi)$ satisfies

$$\rho(x)\frac{dx}{d\xi} = \int_a^b \rho(x)dx, \tag{11}$$

where the right-hand side is independent of ξ. Differentiating again with respect to ξ gives the MPDE,

$$(\rho(x)x'(\xi))' = 0, \quad \text{for } \xi \in \Omega^{[c]}, \tag{12a}$$

whose solution is a mesh generating function, see Definition 2.1. We impose the boundary conditions,

$$x(0) = a \quad \text{and } x(1) = b, \tag{12b}$$

where a and b are the end points of the resulting mesh. The weak form of (12) is: *find, $x(\xi) \in H^1(\Omega)$, such that*

$$\int_0^1 \rho(x)x'(\xi)v'd\xi = 0, \quad \text{for all } v \in H_0^1(\Omega), \quad \text{with } x(0) = a \quad \text{and } x(1) = b. \tag{13}$$

Since (13) is a nonlinear problem, we use a fixed point iterative method to find a solution. At each iteration, k, we solve

$$\sum_{i=1}^{N} \int_{\xi_{i-1}}^{\xi_i} \rho(x_h^{[k-1]})(x_h^{[k]})'\varphi_i' d\xi = 0 \tag{14}$$

The process is repeated until the mesh generated $x_h^{[k]}$ equidistributes $\rho(x_h^{[k]})$; that is, ρ calculated on that mesh.

The definition of a Bakhvalov mesh given in Section 2.2 is essentially that as originally proposed in [3]. Alternatively, this Bakhvalov mesh can be generated by equidistributing the mesh density function

$$\rho(x) = \max\left\{1, K\frac{\beta}{\varepsilon}\exp\left(-\frac{\beta x}{\sigma\varepsilon}\right)\right\}, \tag{15}$$

where K is a positive constant that determines the proportion of mesh points that resolve the layer [11]. When,

$$K\frac{\beta}{\varepsilon}\exp\left(-\frac{\beta\varphi(\tau)}{\sigma\varepsilon}\right) = 1, \tag{16}$$

the transition point, $\varphi(\tau)$, of the mesh generated is the same as in (8). This motivates our choice of K. We insert the calculation for $\chi(\tau)$ in (8) into (16) to get

$$K = \frac{\varepsilon q}{\beta(q-\tau)}.$$

One observes in practice that,

$$\tau \approx q - \frac{\sigma\varepsilon(1-q)}{\beta},$$

so we set

$$K = \frac{q}{\sigma(1-q)}.$$

We can use (16) to generate a Bakhvalov type mesh using an MPDE. When ρ is as defined in (15), a Bakhvalov mesh is a solution to (12), making it a good choice of MPDE for this problem. In Section 2.4 we include algorithms to generate a Bakhvalov type mesh using an MPDE.

Another method to generate a mesh which is similar to a Bakhvalov mesh, and which has a computation advantage over (15), is to equidistribute the mesh density function[13]

$$\rho(x) = 1 + K\frac{\beta}{\varepsilon}\exp\left(-\frac{\beta x}{\sigma\varepsilon}\right). \tag{17}$$

In the layer region the mesh is very similar to a Bakhvalov mesh since the addition of 1 to the value of ρ is insignificant. Outside the layer region the value of the second term is small so the resulting mesh is almost uniform.

2.4 Algorithms

In Algorithm 1 and Algorithm 2 we show how to generate a Bakhvalov type mesh using an MPDE. This is a suitable mesh for the singularly-perturbed problem (6) whose solution has a single boundary layer, near $x = 0$, when the perturbation parameter $\varepsilon \ll 1$. Let $\exp(-\beta s/\varepsilon)$ represent the boundary layer. In Algorithm 3 we show how to generate a two-dimensional layer-adapted mesh using a combination of an MPDE and h-refinement. This mesh is suitable for the two-dimensional singularly-perturbed problem (25) which has layers near $x = 1$ and $y = 1$ which vary in width spatially.

In this section the mesh is defined by the mesh density function,

$$\rho(s) := \max\left\{1, K\frac{\beta}{\varepsilon}\exp\left(-\frac{\beta s}{\sigma\varepsilon}\right)\right\}, \qquad \text{for } s \in \Omega := (0,1). \tag{18}$$

To ensure the mesh generated approximately equidistributes ρ, we calculate the residual of (13) as

$$\text{res}^{[k]} := \sum_{i=1}^{N}\int_{\xi_{i-1}}^{\xi_i} \rho(x_h^{[k]})(x_h^{[k]})'\varphi_i' d\xi, \tag{19}$$

at each iteration k. We stop iterating when $\|\text{res}^{[k]}\|_{0,\Omega} \leq TOL$, an appropriate tolerance level.

[13]RH: I didn't find a reference for this yet.

Niall: Check margins on algorithm boxes. RH: done, I needed to put them in minipages

```
   Input: $N$, the number of intervals in the mesh.
   Input: $\rho$, a mesh density function.
   Input: $TOL$.
 1 Set $\omega^{[c]} := \{\xi_0, \xi_1, \dots, \xi_N\}$ to be a uniform mesh, with $N$ intervals, discretising
     $\overline{\Omega}^{[c]}$;
 2 Set $x(\xi) = \xi$ for $\xi \in \overline{\Omega}^{[c]}$ ;
 3 $s \leftarrow x$;
 4 $k \leftarrow 0$;
 5 do
 6 |   set $x$ to be the $\mathcal{P}_1$-FEM solution, on $\omega^{[c]}$, to
   |
   |       $(\rho(s)x'(\xi))' = 0, \quad \text{for } \xi \in \Omega^{[c]}, \quad \text{with } x(0) = 0, \quad x(1) = 1,$      (20)
   |
   |    $s \leftarrow x$;
 7 |   calculate $\|\text{res}^{[k]}\|_{0,\Omega}$[a];
 8 |   $k \leftarrow k + 1$;
 9 while $\|\text{res}^{[k]}\|_{0,\Omega} > TOL$;
10 Set $\omega^{[p]} := x(\omega^{[c]})$ to be the adapted mesh on $\overline{\Omega}$;
```

Algorithm 1: Generate a layer resolving mesh using an MPDE

[a]Niall: use $\|\text{res}^{[j]}\|_{0,\Omega}$ instead of r. RH: done.

Table 1: Number of iterations to create a Bakhvalov type mesh using Algorithm 1.

$\varepsilon \backslash N$	32	64	128	256	512	1024
1	1	1	1	1	1	1
10^{-1}	4	3	2	1	1	1
10^{-2}	15	14	15	15	14	13
10^{-3}	21	37	56	68	68	70
10^{-4}	21	37	69	132	256	343
10^{-5}	21	37	69	133	260	517
10^{-6}	21	37	69	133	261	517
10^{-7}	21	37	69	133	261	517
10^{-8}	21	37	69	133	261	517

In practice we see that, when using Algorithm 1 to create a Bakhvalov type mesh, the number of iterations required to generate the mesh is $\mathcal{O}(qN)$, where q is the portion of mesh points used to resolve the layer region and N is the number of mesh intervals, see Table 1. This occurs as only one mesh point is added to the layer region at each iteration.

We amend the algorithm by alternating between r- and h-refinement. Initially, we use a uniform mesh with 4 intervals, we refine the mesh three times using r-refinement then double the number of mesh intervals using a uniform h-refinement. In Algorithm 2 we show how to generate a Bakhvalov type mesh using a combination of r- and h-refinement. The number of iterations required on the final mesh are shown in Table 2.

```
Input: $N$, the number of intervals in the final mesh.
Input: $\rho$, a mesh density function.
Input: $TOL$.
1  Set $\omega^{[c;0]} := \{\xi_0, \xi_1, \dots, \xi_4\}$ to be the uniform mesh on $\overline{\Omega}^{[c]}$ with 4 intervals;
2  Set $x(\xi) = \xi$ for $\xi \in \overline{\Omega}^{[c]}$;
3  $s \leftarrow x$;
4  $i, k \leftarrow 0$;
5  while $2^{i+2} < N$ do
6      for $j$ in 1:3 do
7          set $x$ to be the $\mathcal{P}_1$-FEM solution, on $\omega^{[c;i]}$, to
               $(\rho(s)x'(\xi))' = 0, \quad \text{for } \xi \in \Omega^{[c]}, \quad x(0) = 0, \quad x(1) = 1;$     (21)
           $s \leftarrow x$;
8      end
9      $w^{[c;i+1]} \leftarrow$ uniform $h$-refinement of $w^{[c;i]}$;
10     $s \leftarrow s$ interpolated onto $w^{[c;i+1]}$;
11     $i \leftarrow i + 1$;
12 end
13 do
14     set $x$ to be the $\mathcal{P}_1$-FEM solution, on $\omega^{[c;i]}$, to
           $(\rho(s)x'(\xi))' = 0, \quad \text{for } \xi \in \Omega^{[c]}, \quad x(0) = 0, \quad x(1) = 1;$     (22)
       $s \leftarrow x$;
15     calculate $\|\mathrm{res}^{[k]}\|_{0,\Omega}$;
16     $k \leftarrow k + 1$;
17 while $\|\mathrm{res}^{[k]}\|_{0,\Omega} > TOL$;
18 Set $\omega^{[p]} := x(\omega^{[c;i]})$ to be the adapted mesh on $\overline{\Omega}$.;
```

Algorithm 2: Generate a Bakhvalov type mesh using an MPDE, and h-refinement.

An implementation of this algorithm in FEniCS is in Appendix B.

Table 2: Number of iterations to create a Bakhvalov type mesh using Algorithm 2 on final computational domain.

$\varepsilon \backslash N$	32	64	128	256	512	1024
1	1	1	1	1	1	1
10^{-1}	1	1	1	1	1	1
10^{-2}	1	1	1	1	1	1
10^{-3}	1	1	1	1	1	1
10^{-4}	2	1	1	1	1	1
10^{-5}	3	2	2	1	1	1
10^{-6}	3	3	2	2	1	1
10^{-7}	3	3	3	3	2	1
10^{-8}	3	3	3	3	2	2

From Table 2, we can see that using Algorithm 2 reduces the number of iterations required to generate a Bakhvalov type mesh, including the refinements on the coarser meshes, to $\mathcal{O}(\log(N))$.

2.5 Numerical results

In this section we present numerical results for the one-dimensional reaction-diffusion equation (6) and a one-dimensional coupled set of reaction-diffusion equations (23).

2.5.1 A scalar reaction-diffusion equation

We present results for (6) solved on meshes generated using Algorithm 2. When generating the meshes we choose $\sigma = 2.5$, $\beta = 0.99$ and $K = 0.4$ as the values of the variables in our mesh generating function, (18). We set the tolerance level for the L_2-norm of the residual (19) as 0.4.

We can see that the errors in the solutions to (6), measured in the energy norm achieved on the Bakhvalov type mesh, generated using an MPDE, shown in Table 3, converge quadratically, and the bound on the errors is,

$$\|e_h\|_{\mathrm{E},\Omega} \leq C\varepsilon^{1/2}N^{-1},$$

where C is a constant independent of ε and N. These errors are comparable to those generated on a Bakhvalov mesh, using (8), which are shown in Table 4.

Table 3: Errors measured in the energy norm for (6) using Algorithm 2

$\varepsilon\backslash N$	32	64	128	256	512	1024
1	1.10e-05	2.76e-06	6.90e-07	1.73e-07	4.31e-08	1.08e-08
10^{-1}	3.46e-04	8.65e-05	2.16e-05	5.41e-06	1.35e-06	3.38e-07
10^{-2}	2.33e-04	5.81e-05	1.46e-05	3.64e-06	9.09e-07	2.27e-07
10^{-3}	1.21e-04	2.68e-05	5.22e-06	1.29e-06	3.23e-07	8.09e-08
10^{-4}	3.50e-05	8.16e-06	1.91e-06	4.50e-07	1.06e-07	2.59e-08
10^{-5}	1.03e-05	2.54e-06	5.99e-07	1.43e-07	3.42e-08	8.32e-09
10^{-6}	3.27e-06	7.63e-07	1.87e-07	4.46e-08	1.09e-08	2.64e-09
10^{-7}	1.03e-06	2.41e-07	5.77e-08	1.40e-08	3.45e-09	8.51e-10
10^{-8}	3.26e-07	7.62e-08	1.82e-08	4.41e-09	1.09e-09	2.68e-10

Table 4: Errors measured in the energy norm for (6) on a Bakhvalov mesh.

$\varepsilon\backslash N$	32	64	128	256	512	1024
1	1.10e-05	2.76e-06	6.90e-07	1.73e-07	4.31e-08	1.08e-08
10^{-1}	5.73e-04	1.44e-04	3.59e-05	8.97e-06	2.24e-06	5.61e-07
10^{-2}	2.62e-04	6.55e-05	1.64e-05	4.10e-06	1.02e-06	2.56e-07
10^{-3}	9.07e-05	2.16e-05	5.29e-06	1.32e-06	3.29e-07	8.22e-08
10^{-4}	3.12e-05	7.13e-06	1.71e-06	4.21e-07	1.04e-07	2.60e-08
10^{-5}	1.06e-05	2.36e-06	5.54e-07	1.35e-07	3.32e-08	8.26e-09
10^{-6}	3.58e-06	7.76e-07	1.79e-07	4.30e-08	1.06e-08	2.62e-09
10^{-7}	1.20e-06	2.55e-07	5.79e-08	1.38e-08	3.36e-09	8.30e-10
10^{-8}	4.00e-07	8.35e-08	1.87e-08	4.40e-09	1.07e-09	2.63e-10

2.5.2 Coupled system of reaction-diffusion equations

Our second example shows that adapted meshes generated using MPDEs are appropriate for solving problems with solutions that contain sublayers. The chosen problem is a coupled system of two singularly-perturbed reaction-diffusion equations with variable coefficients taken from [12],

$$-\begin{pmatrix}\varepsilon_1 & 0\\ 0 & \varepsilon_2\end{pmatrix}^2\begin{pmatrix}u_1\\ u_2\end{pmatrix}'' + \begin{pmatrix}2(x+1)^2 & -(1+x^3)\\ -2\cos(\pi x/4) & (1+\sqrt{2})\exp(1-x)\end{pmatrix}\begin{pmatrix}u_1\\ u_2\end{pmatrix} = \begin{pmatrix}2x^2\\ 10x+1\end{pmatrix}, \quad \text{for } x\in\Omega := (0,1), \tag{23a}$$

with boundary conditions,

$$u_1(0) = u_1(1) = 0, \quad u_2(0) = u_2(1) = 0. \tag{23b}$$

While both solutions to (23) exhibit layers near $x = 0$ and $x = 1$, only the solution to u_1 has sublayers that are dependant on both ε_1 and ε_2 as can be seen in Figure 5.

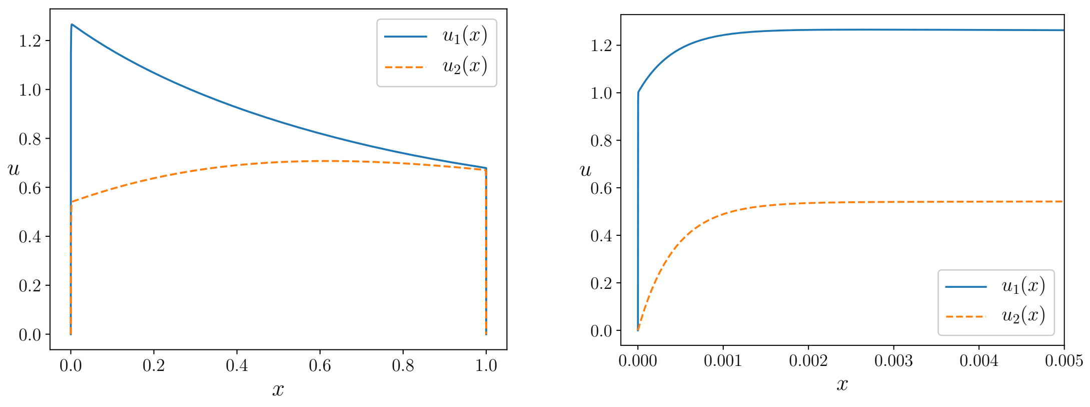


Figure 5: Solutions to (23) with $\varepsilon_1 = 10^{-6}$ and $\varepsilon_2 = 10^{-3}$.

To solve (23) we generate meshes that are graded near $x = 0$ and $x = 1$, in a way that is dependant on both ε_1 and ε_2. To achieve this we solve MPDE (12), using Algorithm 2, with

$$\rho(x) = 1 + K\frac{\beta}{\varepsilon_1}\exp\left(-\frac{\beta x}{\sigma\varepsilon_1}\right) + K\frac{\beta}{\varepsilon_2}\exp\left(-\frac{\beta x}{\sigma\varepsilon_2}\right) + K\frac{\beta}{\varepsilon_1}\exp\left(-\frac{\beta(1-x)}{\sigma\varepsilon_1}\right) + K\frac{\beta}{\varepsilon_2}\exp\left(-\frac{\beta(1-x)}{\sigma\varepsilon_2}\right), \tag{24}$$

with $K = 0.1$, $\beta = 1.99$ and $\sigma = 2.5$. The errors measured in the energy norm $\|E_h\|_{\mathrm{E},\Omega}$ are shown in Table 5, the errors converge linearly, and the bound on the error is

$$\|e_h\|_E \le C\left(\varepsilon_1^{\frac{1}{2}} + \varepsilon_2^{\frac{1}{2}}\right)N^{-1},$$

where C is a constant independent of ε_1, ε_2 and N.

Table 5: Errors in the energy norm for (23) with $\varepsilon_1 = 10^{-6}$.

$\varepsilon_2 \backslash N$	32	64	128	256	512	1024
1	5.29e-02	2.53e-02	1.27e-02	6.34e-03	3.17e-03	1.58e-03
10^{-2}	3.46e-02	1.78e-02	8.96e-03	4.49e-03	2.25e-03	1.12e-03
10^{-4}	2.80e-03	1.45e-03	7.54e-04	3.80e-04	1.91e-04	9.58e-05
10^{-6}	2.95e-04	1.20e-04	5.59e-05	2.76e-05	1.37e-05	6.83e-06
10^{-8}	3.06e-04	1.31e-04	6.29e-05	3.10e-05	1.54e-05	7.71e-06
10^{-10}	3.29e-04	1.43e-04	6.81e-05	3.37e-05	1.67e-05	8.39e-06
10^{-12}	3.30e-04	1.43e-04	6.84e-05	3.39e-05	1.70e-05	8.48e-06

3 Two-dimensional problem

[14] As described in [10][§4.1], a two-dimensional layer-adapted mesh is generated as a uniform mesh in a metric space in $\Omega \subset \mathbb{R}^2$. A matrix function $M = M(\boldsymbol{x})$, the mesh density function, is defined on Ω and the layer-adapted mesh is known as an *M-uniform mesh*. An advantage of a layer-adaptive mesh for singularly-perturbed problems is that the error bound on the layer adapted mesh is dependent on a smaller power of ε and the bound is smaller than if the solution had been generated on a uniform mesh as proven in [10][§2.4]. [15][16]

3.1 Layer-adapted mesh

We used the knowledge gained from generating one-dimensional Bakhvalov meshes using MPDEs to developed algorithms to generate two-dimensional layer-adapted meshes for reaction-diffusion equations using our MPDE approach. We studied problems where diffusion varies on the domain, so the boundary layers vary in width, spatially. Consequently, even when posed on a regular domain (such as a unit square) optimal meshes are not tensor product in nature. Whereas standard Bakhvalov meshes are tensor product, our MPDE approach has no such restriction.

Our chosen two-dimensional singularly-perturbed reaction-diffusion equation, with varying diffusion, is

$$-\nabla \cdot \left(\begin{pmatrix} \varepsilon(1+y)^2 & 0 \\ 0 & \varepsilon(2-x)^2 \end{pmatrix}^2 \nabla u(x,y) \right) + u(x,y) = \left(e^x - 1\right)\left(e^y - 1\right),$$

$$\text{for } (x,y) \in \Omega = (0,1) \times (0,1), \quad (25a)$$

with boundary conditions

$$u = 0 \quad \text{on } \partial\Omega. \quad (25b)$$

The solution to (25) exhibits layers near $x = 1$ and $y = 1$, and a corner layer near $(x,y) = (1,1)$, as can be seen in Figure 6.

[14] Niall: Section title?

[15] RH: Is this sufficient to describe a 2d mesh? NM: Maybe – though a more precise reference is needed. Which section? RH: done.

[16] Niall: Also, compare with [10, §2.4.5]? RH: Done.

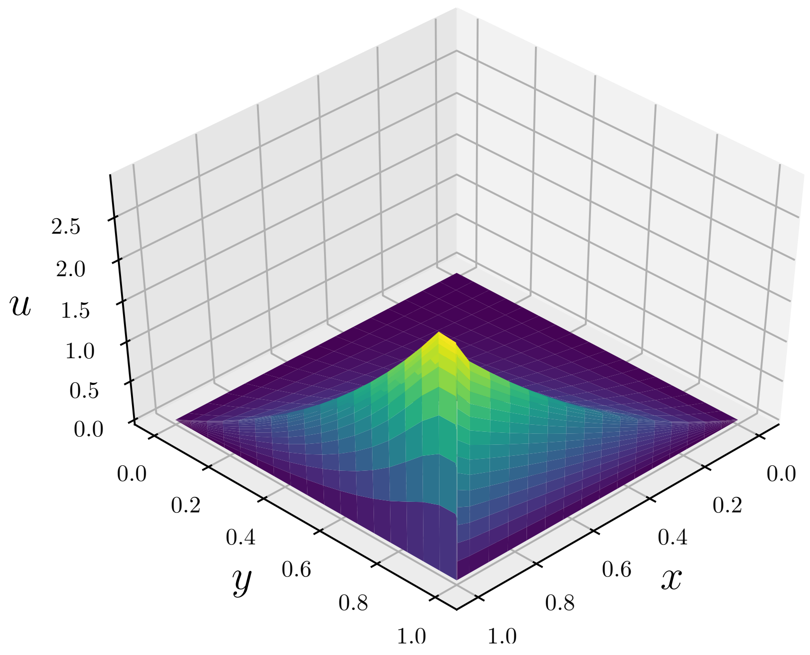


Figure 6: Solution to (25) with $\varepsilon = 10^{-4}$ and $N = 32$.

To generate a layer adapted mesh, that is suitable for (25), using an MPDE we solve the two-dimensional Poisson equation for $\boldsymbol{x}(\xi_1, \xi_2) = (x, y)$, on the domain $\Omega^{[c]} = (0,1) \times (0,1)$,

$$-\nabla \cdot (M(\boldsymbol{x}(\xi_1, \xi_2))\nabla \boldsymbol{x}(\xi_1, \xi_2)) = \boldsymbol{f}(\xi_1, \xi_2), \quad \text{for } (\xi_1, \xi_2) \in \Omega^{[c]}, \tag{26a}$$

with $\boldsymbol{f} = (0,0)^T$, and boundary conditions,

$$\begin{cases} \boldsymbol{x}(0, \xi_2) &= \left(0, \dfrac{\partial \boldsymbol{x}}{\partial \xi_2} = 0\right), \\ \boldsymbol{x}(1, \xi_2) &= \left(1, \dfrac{\partial \boldsymbol{x}}{\partial \xi_2} = 0\right), \\ \boldsymbol{x}(\xi_1, 0) &= \left(\dfrac{\partial \boldsymbol{x}}{\partial \xi_1} = 0, 0\right), \\ \boldsymbol{x}(\xi_1, 1) &= \left(\dfrac{\partial \boldsymbol{x}}{\partial \xi_1} = 0, 1\right). \end{cases} \tag{26b}$$

Here

$$M(\boldsymbol{x}) = \begin{pmatrix} \max\left\{1, K_1 \dfrac{\beta}{\varepsilon_1} \exp\left(-\dfrac{\beta(1-x)}{\sigma\varepsilon_1}\right)\right\} & 0 \\ 0 & \max\left\{1, K_2 \dfrac{\beta}{\varepsilon_2} \exp\left(-\dfrac{\beta(1-y)}{\sigma\varepsilon_2}\right)\right\} \end{pmatrix}, \tag{27}$$

with $\varepsilon_1(x, y) = \varepsilon(1 + y)^2$, $\varepsilon_2(x, y) = \varepsilon(2 + x)^2$, $\beta = 0.99$, $\sigma = 2.5$ and $K_1 = K_2 = 0.4$. An example of the resulting non tensor-product mesh is shown in Figure 7.

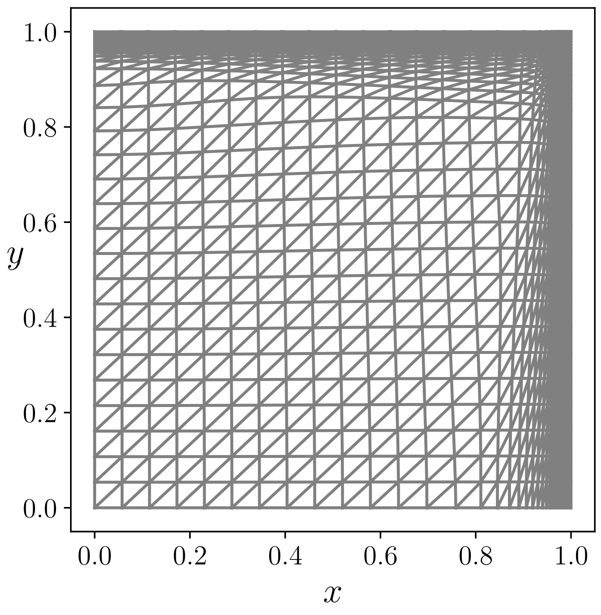


Figure 7: A 32×32 mesh generated using (26) for (25) with $\varepsilon = 10^{-2}$.

3.2 Algorithm

```
Input: $N$, where $N+1$ is the number of mesh points in both directions on $\Omega^{[c]}$.
Input: $M$, a mesh density function.
1  Set $\omega^{[c;0]} := \{(\xi_1,\xi_2)_0, (\xi_1,\xi_2)_1, \dots, (\xi_1,\xi_2)_4\}$ to be the uniform mesh on $\overline{\Omega}^{[c]}$
     with 5 mesh points in each direction on $\overline{\Omega}^{[c]}$;
2  Set $\boldsymbol{x}(\xi_1,\xi_2) = (\xi_1,\xi_2)$ for $(\xi_1,\xi_2) \in \overline{\Omega}^{[c]}$;
3  $\boldsymbol{s} \leftarrow \boldsymbol{x}$;
4  $i \leftarrow 0$;
5  while $2^{i+2} < N$ do
6      for $j$ in $1:3$ do
7          set $\boldsymbol{x}$ to be the $\mathcal{P}_1$-FEM solution, on $\omega^{[c;i]}$, to
               $\nabla \cdot (M(\boldsymbol{s})\nabla \boldsymbol{x}(\xi_1,\xi_2)) = (0,0)^T, \quad \text{for } (\xi_1,\xi_2) \in \Omega^{[c]}, \qquad (28)$
            with boundary conditions as defined in (26b);
8          $\boldsymbol{s} \leftarrow \boldsymbol{x}$;
9      end
10     $w^{[c;i+1]} \leftarrow$ refine $w^{[c;i]}$;
11     $s \leftarrow s$ interpolated onto $w^{[c;i+1]}$;
12     $i \leftarrow i+1$;
13 end
14 for $k$ in $1:4$ do
15     set $\boldsymbol{x}$ to be the $\mathcal{P}_1$-FEM solution, on $\omega^{[c;i]}$, to
           $\nabla \cdot (M(\boldsymbol{s})\nabla \boldsymbol{x}(\xi_1,\xi_2)) = (0,0)^T, \quad \text{for } (\xi_1,\xi_2) \in \Omega^{[c]}, \qquad (29)$
        with boundary conditions as defined in (26b);
16     $\boldsymbol{s} \leftarrow \boldsymbol{x}$;
17 end
18 Set $\omega^{[p]} := x(\omega^{[c;i]})$ to be the adapted mesh on $\overline{\Omega}$;
```

Algorithm 3: Generate a two-dimensional layer-adapted mesh using an MPDE, and h-refinement.

3.3 Numerical results

To solve (25) we generated meshes that are graded near $x = 1$ and $y = 1$, in a way that depends on the varying diffusion coefficient. To achieve this we solve (26) using Algorithm 3 with $M(\boldsymbol{x})$ as shown in (27).

We can see that the errors in the solutions to (25), measured in the energy norm achieved on the layer-adapted mesh, generated using an MPDE, shown in Table 6, converge linearly, and the bound on the errors is,

$$\|e_h\|_{\mathrm{E},\Omega} \leq C\varepsilon^{\frac{1}{2}}N^{-1},$$

where C is a constant independent of ε and N.

Table 6: Errors measured in the energy norm for (25) on a layer-adapted mesh generated using an MPDE.

ε/N	16	32	64	128	256	512
1	2.809e-03	1.457e-03	7.404e-04	3.725e-04	1.866e-04	9.338e-05
10^{-1}	1.663e-02	8.390e-03	4.210e-03	2.108e-03	1.054e-03	5.272e-04
10^{-2}	1.689e-02	8.332e-03	4.153e-03	2.075e-03	1.037e-03	5.185e-04
10^{-3}	8.221e-03	3.790e-03	1.867e-03	9.285e-04	4.640e-04	2.320e-04
10^{-4}	4.728e-03	1.602e-03	6.783e-04	3.221e-04	1.588e-04	7.917e-05
10^{-5}	4.180e-03	1.105e-03	3.301e-04	1.200e-04	5.304e-05	2.562e-05
10^{-6}	4.134e-03	1.044e-03	2.707e-04	7.340e-05	2.295e-05	9.010e-06

4 Conclusion

We have demonstrated how MPDEs can be used to generate *a priori* layer-adapted meshes suitable for one- and two-dimensional singularly-perturbed problems whose solution exhibit layers. These layers include sublayers and layers that vary in width spatially. We combined h- and r-refinement methods to improve the efficiency of the methods. Solutions calculated on these meshes are parameter robust. We've presented algorithms and Python code to enable the reader to generate these meshes in FEniCS.

Appendices

A Python Code to solve (6) on a uniform mesh

[17]

Listing 2: Code to solve (6) on a uniform mesh.

```
## 1DRD_FEM_uniform_mesh.py
"""
Solve the singularly perturbed problem
 -epsilon^2 u'' + u =  f = 1-x, for x in (0,1)
with  u(0) = u(1) = 0
on a uniform mesh

This code is part of
Mesh partial differential equations (PDEs): application and implementation,
Wwitten by Roisin Hill and Niall Madden
Contact: Roisin Hill
For: <cite paper>
"""

from fenics import *

# Define problem parameters
epsilon = 1E-2          # perturbation factor
N = 128                         # mesh intervals

# Create mesh and function space with linear element on that mesh
mesh = UnitIntervalMesh(N)
V = FunctionSpace(mesh, 'P', 1)

u = TrialFunction(V) # Trial functions on V
```

[17]RH: Should I include email addresses in the code?

```
v = TestFunction(V)  # Test functions on V
uN = Function(V)     # Numerical solution in V

# Define the Dirichlet boundary conditions
g = Expression('x[0]*(1-x[0])', degree = 2)
bc = DirichletBC(V, g, 'on_boundary')

# Define right hand side of the SPP, and the weak form
f = Expression ('1-x[0]', degree = 2)
a = epsilon*epsilon*inner(grad(u), grad(v))*dx + inner(u,v)*dx
L = f*v*dx
solve(a==L, uN, bc) # Solve, applying the boundary conditions strongly
```

B Python Code to generate a Bakhvalov type mesh for an SPP whose solution has one layer region near $x = 0$ using a combination of r- and h-refinement.

Listing 3: Code to generate a Bakhvalov mesh using a combination of r- and h-refinement.

```
## 1D_layer_adapted_mesh_rh_refinement.py
"""
Generate 1 sided Bakhvalov type mesh, for a singularly
perturbed ODE whose solution has a layer near x = 0,
by solving the MPDE -(rho(x) x'(xi))' = 0 for xi in (0,1) with
x(0) = 0 and x(1) = 1 and using uniform h-refinements.

Gauss-Lobatto quadrature rule is used:
i.e. replace "return GaussJacobiQuadratureLineRule(ref_el, m)"
with "return GaussLobattoLegendreQuadratureLineRule(ref_el, m)"
in the file FIAT/quadrature.py

This code is part of
Mesh partial differential equations (PDEs): application and implementation,
written by Roisin Hill and Niall Madden
Contact: Roisin Hill
For: <cite paper>
"""

from fenics import *
import math

# Set degree for the Gauss-Lobatto quadrature rule
parameters["form_compiler"]["quadrature_degree"] =  4

# Problem parameters
epsilon = 1E-6          # perturbation factor of the physical PDE
N = 32                          # mesh intervals in final mesh

# parameters for rho(x)
sigma = 2.5                     # related to degree of elements
b = 0.99                        # lower bound on reaction coefficient
q = 0.5                         # proportion of mesh points in layer
K = q/(sigma*(1-q))     # proportion of points in the layer region

# Parameters for h-refinement steps
N_start = 4                     # initial number of mesh intervals
N_step = N_start        # current number of mesh intervals
step_index = 0          # initial step number

# Define right side of PDE and initial solution
f = Expression('0.0', degree = 2)
x_0 = Expression('x[0]', degree = 2)

```

```python
# Define stopping criteria tolerance and intiial value
residual_stop_TOL = 0.2
residual_stop = residual_stop_TOL +1

# Lists for mesh and function space names
mesh_list =  ["mesh0", "mesh1","mesh2","mesh3","mesh4", "mesh5", "mesh6", "mesh7", "mesh8", "mesh9"]
V_list = ["V0","V1","V2","V3","V4","V5","V6","V7","V8","V9"]

# Computational function space parameters
def comp_space(N):
        meshc = UnitIntervalMesh(N)                     # Uniform unit interval mesh
        V = FunctionSpace(meshc, 'P', 1)        # Function space V with linear elements
        v = TestFunction(V)                                     # Test function on V
        x = TrialFunction(V)                            # Trial function on V
        return meshc, V, v, x

# Define weak form of MPDE
def weak_form(rho, xN, x, v, f, meshz, V):
        bc = DirichletBC(V, x_0, "on_boundary") # Define Dirichlet boundary condition
        a = rho(xN)*inner(grad(x),grad(v))*dx(meshz)# Left side of weak_form
        L = f*v*dx                              # Right side of weak_form
        xN = Function(V)                # Numerical solution in V
        solve(a==L, xN, bc)             # Solve, applying the boundary conditions strongly
        return xN

# Define rho(x) for MPDE
def rho(xN):
        return conditional((K*(b/epsilon)*exp(-b*xN/(sigma*epsilon)))>1.0, K*(b/epsilon)*exp(-b*xN/(sigma*epsilon)), 1.0)

# Calculate the norm of the residual to use as stopping criterion
def calc_residual_stop(rho, V, meshN):
        residual = assemble((rho(xN)*xN.dx(0)*v.dx(0)-f*v)*dx) # residual of xN
        bcr = DirichletBC(V, '0.0', "on_boundary")
        bcr.apply(residual)                                     # apply Dirichlet boundary conditions
        residual_func = Function(V)
        residual_func.vector()[:] = residual.get_local() # Residual function on V
        residual_stop = norm(residual_func, 'L2', meshN) # L2-norm of residual
        return residual_stop

# Initial function space parameters
mesh_list[0], V_list[0], v, x = comp_space(N_step) #

# Initial value for xN
xN = interpolate(x_0, V_list[0])

# Iterate through uniform h-refinements
while N_step < N:

        # Calculate solution twice on each mesh size
        for i in range(0,2):
                xN = weak_form(rho, xN, x, v, f, mesh_list[step_index], V_list[step_index])

        # Double the number of mesh intervals
        N_step = N_step*2
        step_index = round(math.log(N_step/N_start,2))

        # Generate computational function space on the finer mesh
        mesh_list[step_index], V_list[step_index], v, x = comp_space(N_step)

        # Interpolate the solution onto the new computational function space
        xN = interpolate(xN, V_list[step_index])

# Calculate solution and L2 norm of the residual on final function space
```

```
while residual_stop > residual_stop_TOL:

        xN = weak_form(rho, xN, x, v, f, mesh_list[step_index], V_list[
            step_index])
        residual_stop = calc_residual_stop(rho, V_list[step_index], mesh_list[
            step_index])

# Generate the physical mesh
meshp = UnitIntervalMesh(N)
meshp.coordinates()[:,0]  = xN.compute_vertex_values()[:]
```

C Python Code to generate a two-dimensional layer-adapted mesh for an SPP whose diffusion varies spatially and has layer regions near $x = 1$ and $y = 1$ using a combination of r- and h-refinement.

Listing 4: Code to generate a two-dimensional layer-adapted mesh using a combination of r- and h-refinement.

```
## 2D_layer_adapted_mesh_rh_refinement.py
"""
Generate a non-tensor product layer adapted mesh, for a singularly
perturbed PDE whose solution has layers that vary spatially near x = 1 and y =
    1,
and a corner layer near (x,y) = (1,1)
by solving the MPDE -grad(M(x) grad(x(xi1, xi2))) = 0 for (xi1,xi2) in (0,1)^2
    with
x(0,xi2) = (0,x_xi2=0),
x(1,xi2) = (1,x_xi2=0),
x(xi1,0) = (x_xi1=0,0),
x(xi1,1) = (x_xi1=0,1),
and using uniform h-refinements.

This code is part of
Mesh partial differential equations (PDEs): application and implementation,
written by Roisin Hill and Niall Madden
Contact: Roisin Hill
For: <cite paper>
"""

from fenics import *
import math

# Problem parameters
epsilon = 1E-2                  # perturbation factor of the physical PDE
N = 32                                  # N+1 mesh points in each direction in
    final mesh

# Parameters for h refinement steps
N_start = 4                             # initial mesh size
N_step = N_start                # current mesh size
step_index = 0                  # initial step index

# Parameters for M(x)
sigma = 2.5                             # dependant on method, often degree of
    elements +1 or +1.5
b = 0.99                                # minimum coefficient of reaction or
    convectio term as appropriate
q = 0.5                                 # proportion of mesh points in layer
K = q/(sigma*(1-q))             # indicates the proportion of points in the
    layer region

# define right side of MPDE
```

```
f = Expression(('0.0','0.0'), degree = 2)

# Lists for mesh and function space names
mesh_list =  ["mesh0", "mesh1","mesh2","mesh3","mesh4", "mesh5", "mesh6", "mesh7", "mesh8", "mesh9"]
V_list = ["V0","V1","V2","V3","V4","V5","V6","V7","V8","V9"]

# generate computational function space
def comp_space(N):
        mesh = UnitSquareMesh(N, N, diagonal = "crossed") # uniform mesh on (0,1)^2
        V = VectorFunctionSpace(mesh, 'P', 1)   # Function space V with linear elements
        v = TrialFunction(V)                            # Test function on V
        x = TestFunction(V)                                     # Trial function on V
        return mesh, V, v, x

# Define boundaries
TOL = 1E-14
def left_boundary( x, on_boundary):
        return  abs(x[0]) < TOL
def right_boundary(x, on_boundary):
        return abs(1-x[0]) < TOL
def bottom_boundary( x, on_boundary):
        return abs(x[1]) < TOL
def top_boundary( x, on_boundary):
        return abs(1-x[1]) < TOL

# Define Dirichlet boundary conditions
def boundary_conditions(V):
        bcl = DirichletBC(V.sub(0), '0.0', left_boundary)
        bcr = DirichletBC(V.sub(0), '1.0', right_boundary)
        bcb = DirichletBC(V.sub(1), '0.0', bottom_boundary)
        bct = DirichletBC(V.sub(1), '1.0', top_boundary)
        bcs = [bcl, bcr, bcb, bct]
        return bcs

# Define the matrix M(x) for MPDE
def M(xN):
        epsilon_y = Expression('epsilon*(1+xi1)*(1+xi1)', epsilon = epsilon, xi1 = xN.sub(0), degree = 2)
        epsilon_x = Expression('epsilon*(2-xi2)*(2-xi2)', epsilon = epsilon, xi2 = xN.sub(1), degree = 2)
        M = Expression((('K*(b/epsilon_x)*exp(-b*(1-xi1)/(sigma*epsilon_x))>1 ? K*(b/epsilon_x)*exp(-b*(1-xi1)/(sigma*epsilon_x)) : 1','0'),\
        ('0','K*(b/epsilon_y)*exp(-b*(1-xi2)/(sigma*epsilon_y))>1 ? K*(b/epsilon_y)*exp(-b*(1-xi2)/(sigma*epsilon_y)) :1')),\
        K=K, b=b,sigma=sigma, epsilon_x = epsilon_x, epsilon_y = epsilon_y, xi1 = xN.sub(0), xi2 = xN.sub(1), degree = 4)
        return M

# Define the weak form
def weak_form(M, xN, x, v, f, bcs, meshz, V):
        a = inner(M(xN)*grad(x), grad(v))*dx(meshz) # Left side of weak weak_form
        L = dot(f,v)*dx                                 # Right side of weak weak_form
        xN = Function(V)                                # Numerical solution in V
        solve(a==L, xN, bcs)                                    # Solve, applying the DirichletBC strongly
        return xN

# Initial function space parameters
mesh_list[0], V_list[0], v, x = comp_space(N_step)

# Set  intial value for xN: xN(xi1, xi2) = (xi1, xi2)
a = inner(grad(x),grad(v))*dx
L = dot(f,v)*dx
xN = Function(V_list[0])
solve(a==L, xN, boundary_conditions(V_list[0]))
```

```

# Iterate through uniform h-refinements
while N_step < N:

        # Generate solution four times on each mesh size
        for i in range (0,4):
                xN = weak_form(M, xN, x, v, f, boundary_conditions(V_list[
                    step_index]), mesh_list[step_index], V_list[step_index])

        # Double the number of mesh intervals in each direction
        N_step = N_step*2
        step_index = round(math.log(N_step/N_start,2))

        # Generate computational function space on the finer mesh
        mesh_list[step_index], V_list[step_index], v, x = comp_space(N_step)

        # Interpolate solution onto the new computational function space
        xN = interpolate(xN,V_list[step_index])

# Calculate solution in final function space
iterations = 0
while iterations < 5:

        xN = weak_form(M, xN, x, v, f, boundary_conditions(V_list[step_index]),
            mesh_list[step_index], V_list[step_index])
        iterations +=1

# Generate the physical mesh
xiX, xiY = xN.split(True)
meshp = UnitSquareMesh(N,N)
meshp.coordinates()[:,0] = xiX.compute_vertex_values()[0:(N+1)*(N+1)]
meshp.coordinates()[:,1] = xiY.compute_vertex_values()[0:(N+1)*(N+1)]
```

References

[1] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J Ring, M. E. Rognes, and G. N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[2] D. Arndt, W. Bangerth, T. C. Clevenger, D. Davydov, M. Fehling, D. Garcia-Sanchez, G. Harper, T. Heister, L. Heltai, M. Kronbichler, R. M. Kynch, M. Maier, J-P. Pelteret, B. Turcksin, and D. Wells. The `deal.II` library, version 9.1. *Journal of Numerical Mathematics*, 2019. accepted.

[3] N. S. Bakhvalov. On the optimization of the methods for solving boundary value problems in the presence of a boundary layer. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 9(4):841–859, 1969.

[4] P. G. Ciarlet. *The finite element method for elliptic problems*. SIAM, 2002.

[5] C. de Boor. Good approximation by splines with variable knots. In *Spline functions and approximation theory*, pages 57–72. Springer, 1973.

[6] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational differential equations*, volume 1. Cambridge University Press, 1996.

[7] M. S. Gockenbach. *Understanding and implementing the finite element method*, volume 97. Siam, 2006.

[8] F. Hecht. New development in FreeFem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[9] W. Huang, Y. Ren, and R. D. Russell. Moving mesh partial differential equations (mmpdes) based on the equidistribution principle. *SIAM Journal on Numerical Analysis*, 31(3):709–730, 1994.

[10] W. Huang and R. D. Russell. *Adaptive moving mesh methods*, volume 174 of *Applied Mathematical Sciences*. Springer, New York, 2011.

[11] T. Linß. *Layer-adapted meshes for reaction-convection-diffusion problems*, volume 1985 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 2010.

[12] T. Linß and N. Madden. Layer-adapted meshes for a linear system of coupled singularly perturbed reaction–diffusion problems. *IMA journal of numerical analysis*, 29(1):109–125, 2008.

[13] K-A. Logg, A. Mardal, G. N. Wells, et al. *Automated solution of differential equations by the finite element method*. Springer, 2012.

[14] MFEM. MFEM: Modular finite element methods library. `mfem.org`, 2019.

[15] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, December 2016.

[16] H-G. Roos. Error estimates for linear finite elements on bakhvalov-type meshes. *Applications of Mathematics*, 51(1):63–72, 2006.

[17] H-G. Roos, M. Stynes, and L. Tobiska. *Robust numerical methods for singularly perturbed differential equations: convection-diffusion-reaction and flow problems*, volume 24. Springer Science & Business Media, 2008.

[18] Y. Shapira. *Solving Pdes in C++: Numerical methods in a unified Object-oriented Approach*, volume 9. SIAM, 2012.

[19] G. I. Shishkin. Grid approximation of singularly perturbed boundary value problems with convective terms. *Russian Journal of Numerical Analysis and Mathematical Modelling*, 5(2):173–187, 1990.