Efficiency and Robustness of an Adaptive Finite Element Method Solver for Some Parabolic Partial Differential Equations

G14SCD

MSc Dissertation in

Scientific Computation

2017/2018

School of Mathematical Sciences

University of Nottingham

Thabo Miles 'Matli

Supervisor: Dr. Kris Van Der Zee

I have read and understood the School and University guidelines on plagiarism. I confirm that this work is my own, apart from the acknowledged references.

Dedication

There once was a mathematician,
who wasn't so good at the written,
but with reiki and dinner,
he was on to a winner,
and free to complete his submission

For my parents

Abstract

We implement the Finite Element Method in space and Finite Differences in time for some variations of the Heat equation. For degrees of freedom n we find convergence of order n^{-2} in the L_2 norm for time steps of size n^{-2}. We then apply space mesh adaptivity to the parabolic problem and run numerical experiments on the adaptive solver comparing tolerance to degrees of freedom. We find evidence to suggest that although the adaptivity is equidistributing effectively the error indicator provides a limited representation of the error distribution. We also investigate object oriented design and find a pattern with separate classes for space meshes, time meshes and tridiagonal matrices to be best.

Contents

1 Introduction

The origin of the Finite Element Method (FEM) is generally agreed to be a paper by Courant [4] in 1943. Though initially obscure, it gained widespread usage in engineering as computing power became more cheaply available. Since then it has become increasingly more common in the natural sciences and more recently in the financial industry [11].

Though more technical than the Finite Difference Method, under certain circumstances the FEM has clear advantages. Two notable advantages are: the FEM is simpler to use for Partial Differential Equations (PDEs) with irregular shaped domains, and that there is a very well understood theory of aposteriori errors. This theory of errors, which only requires knowledge of the estimated solution, allows for the FEM to be adapted during implementation.

The pioneering work of Babuska et al [2] in the 1980s showed the first examples of how an aposteriori error estimate could be used to implement adaptive FEM. The research moved quickly and attempts at adaptive mesh refinement for parabolic PDEs began towards the end of the same decade see [5], [6] and others. Despite the research into these methods and the use of adaptive FEM in science and engineering there are still some theoretical results outstanding. Convergence and optimal complexity have only been shown for linear elliptical PDEs and only quite recently [8], [10]. Even these recent results only show that there is convergence to a solution and do not imply an order of convergence for the adaptive methods.

This dissertation continues that investigation into the characteristics and outcomes of the adaptive FEM focussing on parabolic PDEs. Adaptivity leads to a solution that should be in some sense efficient as ideally maximum accuracy would be achieved for the minimum degrees of freedom. It is this proposition that we investigate in this dissertation and address this question directly with experiments in section 7.1.

Another aim of this dissertation is the creation of a code which is safe and scalable. FEM software packages are available with increasing quality and functionality. Scientific software which is to be shared publicly or commercially needs to meet high standards of design and engineering. This will be done through object orientation as explained in [9] and other similar works. Throughout this dissertation we analyse the design choices

taken and the data structures created to meet this goal.

These are ambitious goals for one dissertation. To do this we will briefly introduce the PDE under consideration which is the Heat Equation in section 2. We will then introduce some background to the FEM and solve an elliptical problem in section 3, then do the same for the parabolic Heat Equation in section 4. In section 5 we will introduce our a posteriori error indicator and create an adaptive code. Then in the final section we will investigate numerically the concept of efficiency in adaptive FEM solvers.

All codes in this dissertation are built from standard C++14. The only external code used is for Guass-Legendre quadrature which comes from the Boost library. Further details of this and other excellent peer reviewed scientific codes can be found on the boost website `https://www.boost.org/doc/libs/1_66_0/libs/math/doc/html/math_toolkit/gauss.html`. Anyone looking to compile the code in this dissertation will have to install this package.

2 The Linear Heat Equation

The Heat Equation provides a simple situation within which we can understand and demonstrate the implementation of the Finite Element Method and adaptive algorithms. Conversely it is fundamental enough that it would allow someone reading this dissertation to easily build on our findings to access a different but connected problem for example the Black-Scholes equation. It can be thought of as a prototypical parabolic equation and this is our motivation for studying it.

In this section we will describe the derivation of the Heat Equation in one dimension. We will first do this in the steady state i.e. where $\frac{du}{dt} = 0$ and then extend this to the time dependent equation. We do this just to refresh the reader's knowledge of the properties of the equation.

2.1 The Heat Equation in Steady State

The steady state equation in one dimension can be thought of as describing a thin rod of uniform material on the interval $I = [0, \mathrm{L}]$. As we are only considering the one dimensional problem there is only diffusion in the x direction. The rod is heated by a source f which we assume to have been acting continuously for long enough to reach the steady state.

Let q be the heat flux i.e flow of energy per unit of area per unit of time and let S be the cross section of the rod. As flux is a vector quantity it needs a direction and we take this as the direction of x increasing. The first law of thermodynamics on conservation of energy tells us that the flow out of the rod must equal the flow in from the heat source. Hence we have:

$$q(L)S(L) - q(0)S(0) = \int_I f dx \tag{2.1}$$

We now divide both sides of (2.1) by L and take $\mathrm{L} \to 0$. Hence we have the differential equation:

$$(Sq)' = f \tag{2.2}$$

Employing Fouriers law which in this context can be understood as the flux being

negatively proportional to the temperature gradient:

$$q = kT' \tag{2.3}$$

here k represents the heat conductivity of the rod.

Together (2.2) and (2.3) give us the Heat Equation:

$$-(SkT')' = f \tag{2.4}$$

2.2 The Transient Heat Equation

The time dependent problem is very similar to the steady state form. We introduce a function e which is the energy per unit length within the rod. We use the the conservation of energy principle again but this time we use the fact that the sum of the resultant heat flux is equal to the rate of change of internal energy. Here rate of change of e is denoted by $\dot{e}$

$$\int_I \dot{e}dx = q(0)S(0) - q(L)S(L) \int_I f dx \tag{2.5}$$

Once again dividing by L and letting L $\to$ 0.

$$\dot{e} + (Sq)' = f \tag{2.6}$$

Finally assuming that the energy depends linearly on the Temperature

$$e = mT \tag{2.7}$$

And using (2.3) and combining (2.5) and (2.6) we have the transient Heat Equation which we will refer throughout the rest of this dissertation simply as the Heat Equation:

$$m\dot{T} - (SkT')' = f \tag{2.8}$$

3 The Finite Element Method

This section is concerned with the discretisation of the elliptical Heat Equation by Finite Elements. Although the focus of this dissertation is adaptive solvers for the transient Heat Equation many aspects of the time dependent problem very similar in the stationary form. We will therefore introduce a bulk of concepts in this chapter to avoid introducing this content alongside the specific content for the parabolic problem.

Though not uncommon, the FEM may not be known by the average postgraduate mathematician. As such we will provide a fairly detailed description of the method and a few of its attendant concepts. If more detail is needed a nice introduction to the topic is available in [7].

The underlying practical steps to the method can be reduced to finding a variational form of the equation and then solving that form of the equation in an approximated sense. We will add more detail as we introduce the relevant background however the steps to a Finite Element Method can be loosely grouped into the following steps:

1. Finding a variational form of the equation we are looking to solve.
2. Creating the subdivision of our domain Ω.
3. Taking the variational form, which we have at first defined in an infinite dimensional function space V, and approximating it on a finite dimensional sub-space V_h. In our case the subspace will be piecewise polynomial functions defined on our subdivision.
4. Using an ansatz for u_h involving our chosen basis function.
5. Solving the resulting system of equations

The remainder of this chapter will introduce some function spaces and the basis function that we will use for our approximation 3.1, we will then introduce the weak formulation and approximation of an elliptical model problem 3.2, an analysis of the model problem will be carried out in 3.3 and then finally we will analyse the way we have designed our data structures and why in 3.4.

Once the mathematical context of the Finite Element has been outlined we will move on to finding the weak formulation of the Heat Equation. With this done we will consider

the finite dimensional subspace and the elements that will be used in our discretisation. We will close the chapter will a discussion of other possible choices for elements.

3.1 Supporting Theory

We begin by stating the notation to be used in this dissertation and also stating some of the relevant definitions and results from functional analysis.

3.1.1 Notation and the Weak Derivative

Definition 3.1. Continuous Functions Let $\Omega \subset \mathbb{R}^n, n \geq 1$ be an open bounded set.

- We define $C(\Omega)$ the set of all real-valued and continuous functions defined on Ω.
- For $m \geq 1$ we write $C^m(\Omega)$ to denote the set of m times continuously differentiable functions i.e. that $C^m(\Omega) = \{f \in C(\Omega) : f^{(k)} \in C(\Omega) \quad \forall k \leq m\}$.
- Let $C_0^\infty(\Omega)$ denote the set of all infinitely many times differentiable that vanish on the boundary of Ω.

The multi-index is a notational tool which allows for the simplification of statements in multi-dimensional calculus and is useful for our descriptions of PDEs. It is specifically an array α which carries information about our partial derivatives.

Definition 3.2. Multi Index

$\alpha = (\alpha_1, \alpha_2, ...\alpha_n) \in \mathbb{N}^n \qquad |\alpha| = \alpha_1 + ...\alpha_n$
$D^\alpha = \left(\frac{\partial}{\partial x_1}\right)^{\alpha 1} ... \left(\frac{\partial}{\partial x_n}\right)^{\alpha_n} = \frac{\partial^{|\alpha|}}{\partial x_1^\alpha ... \partial x_n^\alpha}$
$C^m(\Omega) = \{f \in C(\Omega) : D^\alpha f \in C(\Omega) \quad \forall |a| \leq\leq m\}$

The FEM makes extensive use of piecewise-linear functions. These functions may not have derivatives in the traditional continuous sense but we may generalise the concept of a derivative to admit certain piecewise functions.

Definition 3.3. Weak Derivative

Let u be a locally integrable function on Ω. Say there exists a function ψ_α which is also locally integrable on ω such that:

$$\int_\Omega \psi_\alpha(x) \cdot v(x)dx = (-1)^\alpha \int_\Omega u \cdot D^\alpha v \forall v \in C_0^\infty(\Omega)$$

Then ψ_α is the weak derivative of u of order $|a|$ hence $\psi_\alpha = D^\alpha u$.

3.1.2 Function Spaces

To find a suitable weak formulation we need to take some results from the theory of function spaces.

Definition 3.4. L_2 Space Let $L_2(\Omega)$ denote the set of all real valued functions defined on Ω with $\Omega \subset \mathbb{R}^n$ such that:

$$\|u\|_{L_2(\Omega)} := \Big(\int_\Omega |u|^2 dx \Big)^{\frac{1}{2}} < \infty$$

Definition 3.5. Sobolev Space

$H^m(\Omega) = \{u \in L_2(\Omega) : D^\alpha u \in L_2(\Omega), |\alpha| \leq m\}$

$$\|u\|_{H^m(\Omega)} := \Big(\sum_{|\alpha| \leq m} \|D^\alpha u\|^2_{L_2(\Omega)} \Big)^{\frac{1}{2}}$$

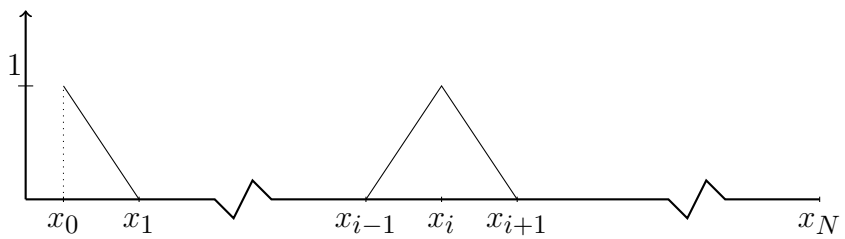
3.1.3 Basis Functions

Take the interval $x \in [0, L]$ and the intervals $[x_k, x_{k+1}] \quad k = 0, 1, ..., N-1$ with $h_i = x_{i+1} - x_i$. Then we can define the below function which is referred to a the FEM basis function or the "hat" function.

$$\phi_j(x) = \Big(1 - \Big| \frac{x - x_k}{h_i} \Big| \Big)_+, \quad j = 0, 1, ..., N \tag{3.1}$$

The interval leads to two half hats on the boundary. Clearly the basis function does not have a continuous derivative but it does have a weak derivative in the sense described above 3.3.

The function space through which we will choose to view this function



3.2 An Elliptical Model Problem

Now we have defined the mathematical context let us solve a problem via the FEM. Take the stationary heat equation. The existence and uniqueness of a solution for this problem follows from the Lax Milgram theorem given in appendix A.

$$-u'' = f \qquad x \in [0,1] \tag{3.2a}$$

$$u(0) = 0 \tag{3.2b}$$

$$u(1) = 0 \tag{3.2c}$$

The first step of solving this and any PDE by the FEM is to find a suitable weak formulation. This is given by multiplying both sides of the equation by a test function in the Sobolev space $v \in H_0^1(0, L)$.

$$\int_0^L u''v dx = \int_0^L fv dx \qquad dx \forall v \in H_0^1(0, L)$$

Integrating by parts on the left leads us to our weak formulation of 3.2

Problem 3.1. *Weak Formulation*

Find $u \in H_0^1(0, 1)$*:*

$$\int_0^L u'v' dx = \int_0^L fv dx \qquad \forall v \in H_0^1(0, L)$$

$\forall v \in H_0^1(0, L)$

Define

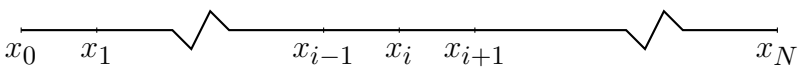
$$\int_0^L u'v' dx = a(u, v) \int_0^L fv dx = \ell(v)$$

This gives us the linear and bilinear form with can be used to show by A that u exists and is unique. However, our test function has been chosen simply some function in the Sobolev space which has infinitely many functions defined within it and is described as infinite dimensional.

To find the FEM approximation to 3.2 we look for a solution in a finite dimensional subspace of $V \subset H_0^1(0, L)$. The subspace we chose consists of continuous piecewise linear

functions of defined on some given subdivision of Ω. Then we can restate the weak formulation by the following:

Take the subdivision on $x \in \Omega$ in our model problem 3.2. We subdivide the domain into N intervals $[x_k, x_{k+1}] \quad k = 0, 1, ..., N-1$. This subdivision may not be regular and when later in this dissertation we begin seeking an adaptive solution will very rarely be regular. intervals may be referred to as elements and give the FEM its name.



Problem 3.2. *FEM Approximation Find* $\quad u_h \in V \subset H_0^1(0,1) \quad :$
$a(u_h, v_h) = \ell(v_h) \quad \forall v_h \subset V.$

Now we can define our approximation u_h to 3.2 by the ansatz:

$$u_h(x) = \sum_{j=1}^{N(h)} U_j \cdot \phi_h(x) \tag{3.3}$$

With $U_j \; j = 1, ..., N(h)$ being constants to be determined. We can now rewrite 3.2 by the following:

$$\sum_{j=1}^{N(h)} U_j \; a(\, \phi_j(x),\, \phi_i(x)) = \ell(\, \phi_i(x)\,)$$

This gives a system of linear equations with $a(\, \phi_j(x),\, \phi_i(x))$ an $N(h) \times N(h)$ matrix.

$$A_{ij} = \int_I \phi_i' \phi_j' dx \tag{3.4}$$

$$F_i = \int_I f \phi_i dx \tag{3.5}$$

Hence the system $AU = F$.

3.3 Numerical Experiments

First we take 3.2 a uniform partition with interval size h such that.

$$-u'' = \pi^2 \cos(\pi x - 0.5\pi) \qquad x \in [0,1] \tag{3.6a}$$

$$u(0) = u(1) = 0 \tag{3.6b}$$

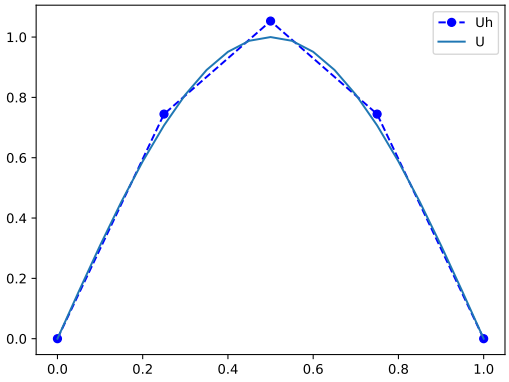
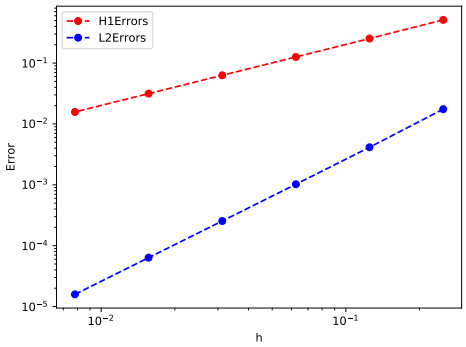


Figure 1: An approximation for constant h

h	$\|u-u_h\|_{L_2(0,1)}$	$\|u-u_h\|_{H^1(0,1)}$
$\frac{1}{4}$	0.0174706	0.511854
$\frac{1}{8}$	0.00413879	0.252837
$\frac{1}{16}$	0.00102063	0.12604
$\frac{1}{32}$	0.000254283	0.0629727
$\frac{1}{64}$	6.3516e-005	0.0314805



We see smooth convergence in both of our chosen norms. In $\|u-u_h\|_{L_2(0,1)}$ the order of convergence is approximately h^2 and in $\|u-u_h\|_{H_0^1(0,1)}$ is approximately h. This tells us that the second term in $\|\cdot\|_{H_0^1(0,1)}$ related to the derivative is dominating and is order h^2.

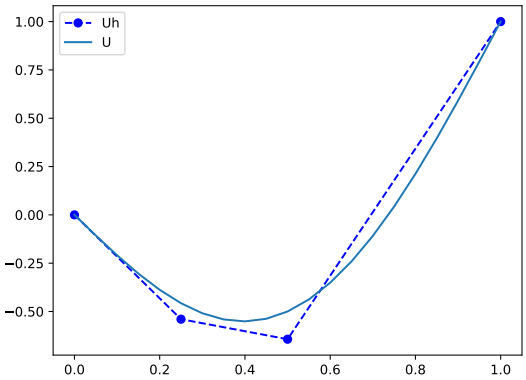
This is a result that those with a knowledge of FEM may have been expecting. As we begin designing and investigating adaptive algorithms we will have to consider non-uniform meshes. The concept of interval width will remain important but from now on we also consider degrees of freedom (DOFs)which is the number of nodes in a given partition

including the boundary nodes. Take following adjustment of 3.2 and its approximation without uniform intervals.

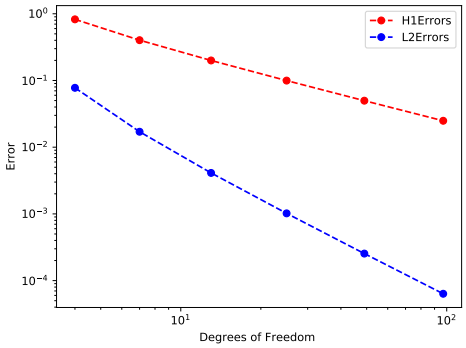
$$-u'' = \pi^2 \sin(\pi x) \qquad x \in [0,1] \tag{3.7a}$$

$$u(0) = 0 \tag{3.7b}$$

$$u(1) = 1 \tag{3.7c}$$



DOFs	$\|u - u_h\|_{L_2(0,1)}$	$\|u - u_h\|_{H^1(0,1)}$
4	0.0776354	0.825144
7	0.0170398	0.40185
13	0.00411272	0.199548
25	0.00101902	0.0996014
49	0.000254182	0.0497791



We see the hoped for convergence with respect to the degrees of freedom. As we perhaps expected we see the order of convergence in the L_2 and H_1 norms of n^{-2} and n^{-1} respectively where n represents the degrees of freedom.

3.4 Implementation

It is evident that all the matrices that arise from the one dimensional FEM are tridiagonal. This is important as there are efficient algorithms available for solving tridiagonal systems.

This is an ideal opportunity to create a data structure to store these matrices and associate the any methods that we would like to use throughout the project. To do this we define our base class for tridiagonal matrices and define within it a method "Matrix Solver" which implements the Thomas algorithm. One possible declaration of this design pattern is given below and the full implementation is given in appendix B:

```
class TriDiagMatrix{
public:
    void MatrixSolver( std::vector<double> f,
         std::vector<double> &x );

protected:
    std::vector<double> mpDiagonal;
    std::vector<double> mpLowerDiag;
    std::vector<double> mpUpperDiag;
};
```

The obvious advantage of a similar declaration is not only the efficiency of the storage and the methods but also the readability, polymorphism and the ease with which the code can then be tested. Once module tests have been run and effectiveness established we can derive any specific matrices we need from the above base. The first derivation we make is for 3.4 also called the Stiffness matrix. This class reads a specific subdivision of the domain and sets the Stiffness matrix accordingly. The program files for the Stiffness matrix is given in appendix C

```
class StiffnessMatrix: public TriDiagMatrix{
public:
    void BuildStiffnessMatrix ( SpaceMesh& smesh ); };
```

With these tools we can present the elliptic solver. The major point we emphasise here is readability and code safety. As we move to solving parabolic PDEs in the next section this modular approach will allow our code to scale.

```
void StationaryHeatEquation(){
```

```
StiffnessMatrix ( mpsmesh );
buildfvec( mpsmesh)
stiff.MatrixSolver( f_vec, U );
PrintVector(U);
}
```

3.5 Boundary Conditions

We have not yet discussed in detail the implementation of boundary conditions. This has been done in 3.1 by choosing $H_0^1(0, L)$ for our homogeneous dirichlet boundaries. In general we use this "trick" in this dissertation when defining the weak formulation. This is done to avoid delving into the detail of choosing the correct solution space for our solution which is outside of the remit of this dissertation.

However, in practise one of the major advantages of the FEM is the ease with which more complicated boundary conditions can be implemented. One method for doing this is a penalty method of implementing Robin boundary conditions and using parameters to approximate non-homogeneous dirichlet conditions. Neumann boundaries can also be implemented in this way.

A detailed account of implementing Robin boundary conditions with parameters is given in [7]. Where our examples extend to PDEs with non-homogeneous boundary conditions this is what we have done.

4 Discretising the Time Dependent Problem

Now that we have built some foundations by solving an elliptical PDE refeq:Steady Diffusion1 we can consider parabolic Heat Equation. The steps involved in solving the parabolic version are:

1. We discretise in space as before, however we adjust the solution space to consider our basis functions for a sequence of specific times that vary in x. This can be thought of loosely as a sequence of elliptical problems.

2. Creating the subdivision of our domain Ω in both space and time. Replace our infinite dimensional function space by a finite dimensional subspace. We see that we can use the same basis functions.

3. We discretize in time using a Finite Difference method -in our case the Implicit Euler method.

4. We solve the resulting system of equations at the given time steps

Once we have carried out these steps we will perform some numerical experiments and convergence analysis and then close the chapter with a discussion of mesh design and related data structure. "Meshing" is the art-form that underpins the adaptive FEM and so merits close attention.

4.1 Semi-discrete Problem

To describe the semi-discrete problem we will take the following model problem.

$$\dot{u} - u'' = f \quad x \in [0, L], \quad t \in (0, T] \tag{4.1a}$$

$$u(0, t) = u(L, t) = 0 \tag{4.1b}$$

$$u(x, 0) = u_0(x) \tag{4.1c}$$

Multiplying $\dot{u} - u'' = f$ by a test function $v \in H_0^1(0, L)$ on both sides.

$$\int_0^L \dot{u}v dx - \int_0^L u''v dx = \int_0^L fv dx$$

Integrating by parts yields:

$$\int_0^L \dot{u}v dx + \int_0^L u''v + u(0)v(0) - u(L)v(L)dx = \int_0^L fv dx$$

$$\int_0^L \dot{u}v dx + \int_0^L u'v' dx = \int_0^L fv dx$$

The weak formulation of our problem is now given by:

Problem 4.1. *Weak Formulation*

Find $u(\cdot, t) \in H_0^1(0, L)$*:*

$$(\dot{u}(\cdot, t), v) + a(u(\cdot, t), v) = (f(\cdot, t), v)$$

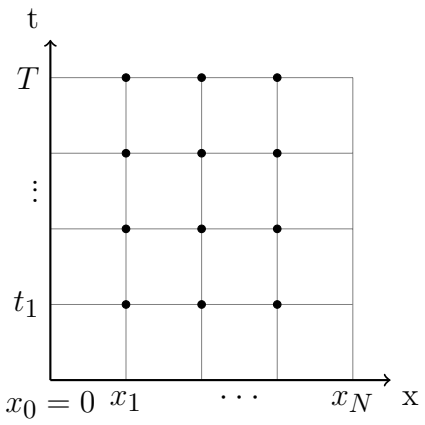
$$(u(\cdot, 0), v) = (u_0, v)$$

$\forall v \in H_0^1(0, L)$

Here,

$$a(w, v) = \int_0^L w'v' dx (w, v) = \int_0^L wv dx$$

To find our approximation define a mesh on $[0, 1] \times [0, T]$.



Let the partition on the space mesh be $0 = x_0 < x_1 < ... < x_N = 1$ with intervals $h_i = x_i - x_{i-1}$. Let the time partition be $0 = t_0 < t_1 < ... < t_M = T$ time steps $\Delta t_j = t_j - t_{j-1}$. For both space and time we allow the intervals to vary.

Let $V \subset H_0^1(0,1)$ denote the set of all continuous piecewise linear functions defined on the space mesh that are zero on the boundaries $x = 0$ and $x = 1$.

4.1.1 Implicit Euler

Problem 4.1 is now discretised in space this is sometimes referred to as the semi-discreet problem. We now need to discretise in time to be able to find an approximation to the solution. We will use Finite Differences to discretise in time and we will choose the Implicit Euler method.

Problem 4.2. *Implict Euler Method*

Find $u_h^{m+1} \in V \quad 0 \leq m \leq M-1$

$$\begin{aligned} \Big(\frac{u_h^{m+1} - u_h^m}{\Delta t}, v_h\Big) + a(u_h^{m+1}, v_h) &= (f(\cdot, t^m), v_h) \\ (u_h^0 - u_0, v_h) &= 0 \quad \forall v_h \in V \end{aligned}$$

We can therefore rewrite the system as:

$$\begin{aligned} (u_h^{m+1},\, v_h) + \Delta t a(u_h^{m+1}, v_h) &= (u_h^m,\, v_h) + \Delta t (f(\cdot, t^m), v_h) \\ &\forall v_h \in V \quad 0 \leq m \leq M-1 \end{aligned}$$

with

$$(u_h^0) = (u_0, v_h) \quad \forall v_h \in V$$

Then as in section 3 we choose the FEM basis function and use the ansatz 3.3 to form a system of equations at each time. It may be seen that the term originating from the rate of change gives rise to a new matrix sometimes called the Mass matrix. A class of Mass matrices is derived from the Tridiagonal matrix base in appendix D.

Also as we now have multiple PDEs it is helpful to create an interface to deal with them and pass them in general to a solver class. The solver will expect an abstract class with all the possible characteristics that any of our PDEs could have. We can then override the relevant virtual functions to create a specific PDE. An abstract class to form the interface

is given in appendix G and a specific example of a PDE for one of our parabolic model problems is given in appendix H.

4.2 Numerical Experiments

4.2.1 A Convergence Study

Take the model problem 4.1b without a source function and thermal diffusivity π^{-2}

$$\dot{u} - \pi^{-2}\, u'' = 0 \quad x \in [0, L], \quad t \in (0, 1] \tag{4.2a}$$

$$u(0, t) = u(1, t) = 0 \tag{4.2b}$$

$$u(x, 0) = 6sin(\pi x) \tag{4.2c}$$

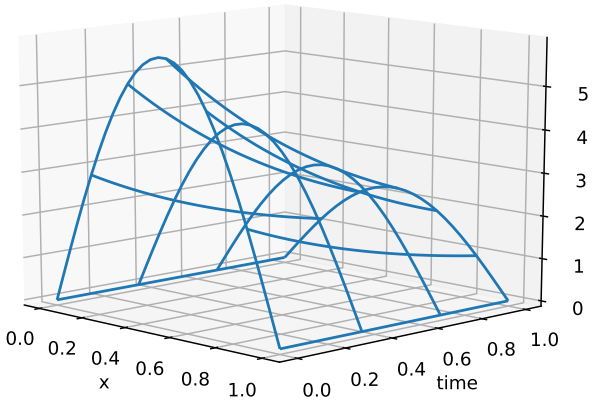


Figure 2: The analytic solution to 4.2a shows the diffusion of heat over time in our system

To study the convergence of our solution we set $\Delta t = n^{-2}$ for time step of size Δt and degrees of freedom n. We then study the solution at the final time for various numbers of degrees of freedom.

We have convergence of approximately order n^{-2} in the L_2 norm and order n^{-1} in the H^1. This is perhaps what we expected from the stationary example however the choice $\Delta t = n^{-2}$ was important to find this convergence.

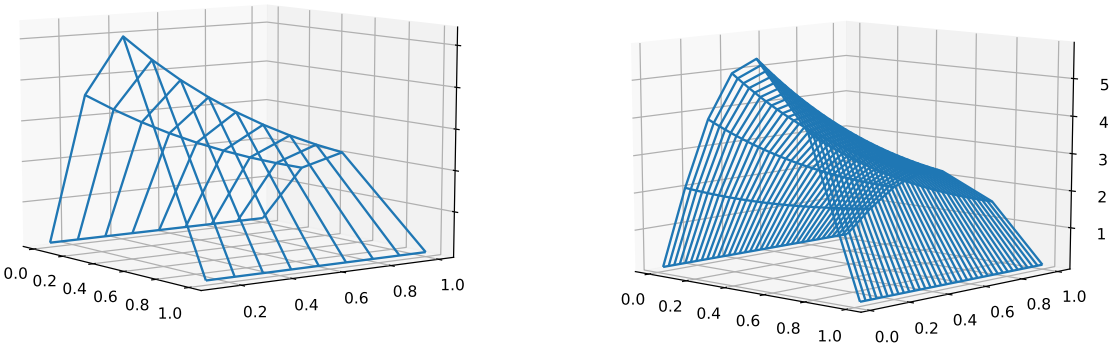


Figure 3: The plots of our approximation appear to follow our analytic solution 2

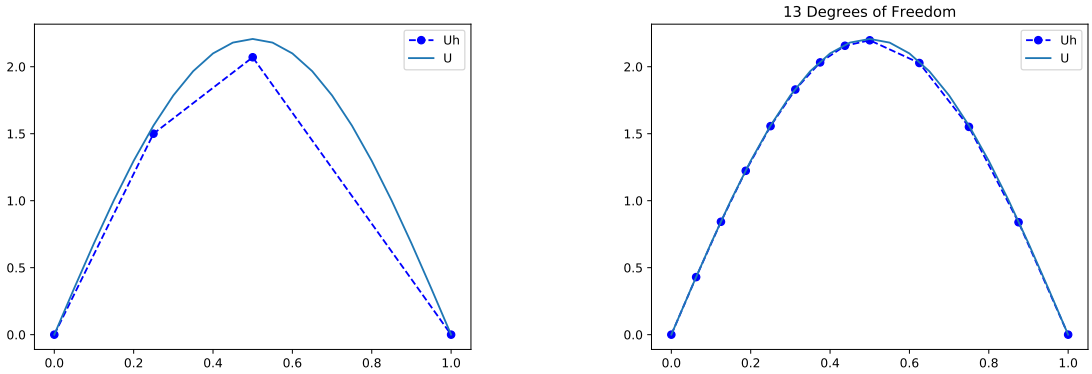
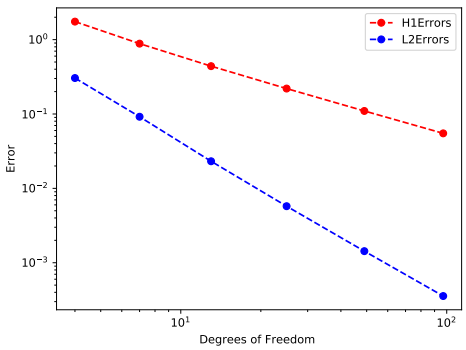


Figure 4: We can see the approximation converging to the true solution in the final time

DOFs	$\|u - u_h\|_{L_2(0,1)}$	$\|u - u_h\|_{H^1(0,1)}$
4	0.304466	1.74657
7	0.0918245	0.883062
13	0.0231464	0.439949
25	0.00575675	0.219779
49	0.00143142	0.109867



4.2.2 Another Convergence Study

We will now justify this choice numerically by looking at convergence for a different choice of Δt. Consider again the model problem 4.1b with adjusted boundary conditions and initial data. This time set $\Delta t = n^{-1}$.

$$\dot{u} - \pi^{-2}\, u'' = 0 \quad x \in [0, L], \quad t \in (0, 1] \tag{4.3a}$$

$$u(0, t) = 0 \quad u(1, t) = 1 \tag{4.3b}$$

$$u(x, 0) = 2sin(2\pi x) + x \tag{4.3c}$$

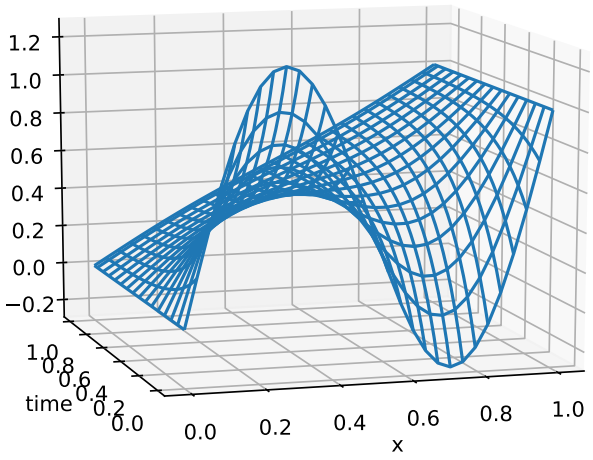


Figure 5: The analytic solution to 4.3a shows the diffusion of heat over time in our system

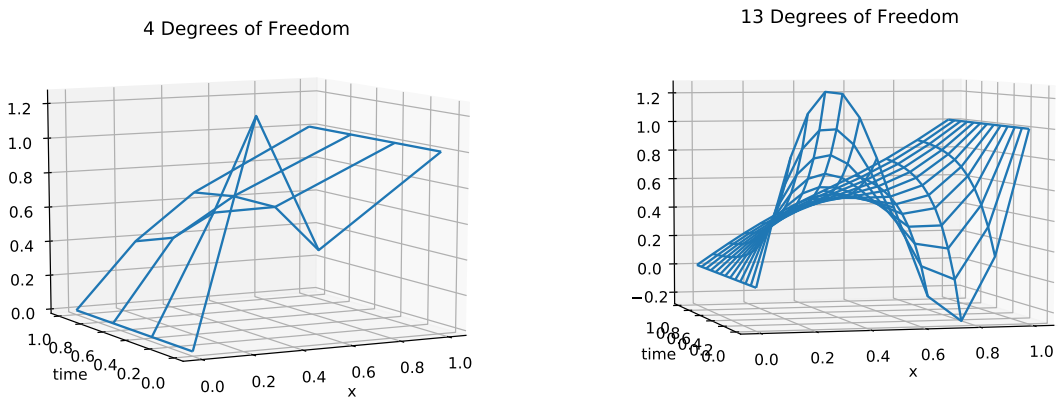


Figure 6: We see that the coarse approximation is quite different from figure 4.3a which is a symptom of the mesh we have used. However, the fine approximation begins to approach the expected shape

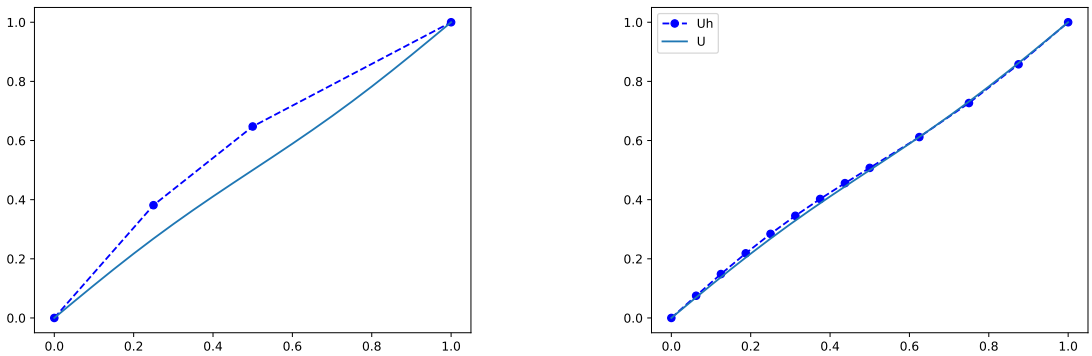


Figure 7: We have the appearance of convergence in the final time as well

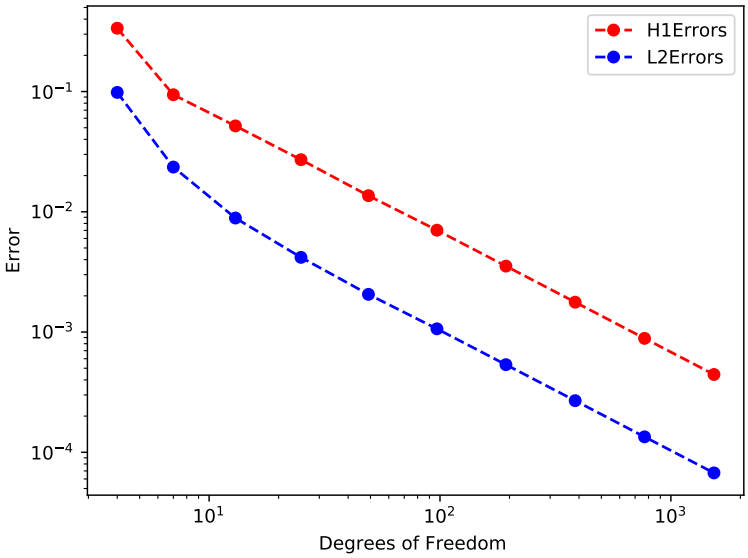


Figure 8: When we explore the approximations in the L_2 and the H_1 norms we see some limitations of our choice $\Delta t = n^{-1}$. This is because both the FEM and the Implicit Euler method incur errors. The FEM converges order L_2 and H_1 norms of n^{-2} and n^{-1} as we showed in subsection 3.3. The implicit Euler converges order Δt so the convergence of the fully discreet problem is order $\Delta t + n^{-1}$ in the L_2 norm and $\Delta t + n^{-2}$ in the H^1 norm. We can therefore see that for the choice $\Delta t = n^{-1}$ the term Δt limits the order of convergence. This is undesirable as in this dissertation we are primarily interested in the efficiency of the adaptive algorithm with respect to degrees of freedom. From now on in this dissertation we set $\Delta t = n^{-2}$ as in subsection 4.2.2 so that we can retain a focus on the quality of the solution wrt to DOFs.

4.3 Implementation

"Meshing" is the important art-form on which the computer implementation of the FEM depends. There are entire software packages available to carry out the creation and manipulation of mesh structures and it would therefore benefit any FEM project to either design there own data-structures or implement an existing mesh package.

An important question when designing the data-structure and class inheritance structure is the relationship between the time-steps and the space nodes. In our space nodes are free to vary at a given time step however the time steps are the same when considered from a given node. This seems to both invite and discourage the creation of a single base class from which to derive meshing in space and time. Much of the functionality of each is shared but much differs:

```
class SpaceMesh
{
public:
    void CopySpaceMesh (const SpaceMesh& oldSpaceMesh);
    void BisectIntervals (std::vector<int> &intervalsForBisection);
    void CommonMesh( SpaceMesh& firstmesh, SpaceMesh& secondmesh );
    double TestFunctions(int nodeIndex, double x);
    bool Contained (double my_var );
protected:
    std::vector<double> mpSpaceNodes ;
};

class TimeMesh
{
public:
    void CopyTimeMesh (const TimeMesh& oldTimeMesh);
    void GenerateTimeMesh ( std::vector<double> TimeNodes );
    void BisectInterval (int lowerIndex, int upperIndex);
protected:
    std::vector<double> mpTimeNodes;
};
```

The question of how to structure inheritance in FEM meshes is in this case restricted to the one dimension and may seem abstract at the beginning of a project. However, there are many utility style functions i.e. adding a node or copy construction which could be simplified if there was an abstraction which could be justified rigorously. The danger of a heuristic abstraction pattern is that one of the child classes have access to a function which cannot be used safely or that the program not scale because of access restriction to prevent unsafe function calls. Though we found it prudent to separate time and space meshes some future work may under cover a good design pattern to combine them under a base class. The program file for some of the important methods in SpaceMesh class are included in appendix E. The full header file for TimeMesh is also included for consideration in appendix F. However, he program files are not included as most are just combinations of fairly simple algorithms performed on dynamic arrays.

5 A Posteriori Error Analysis

When a continuous problem is discretised by Finite Element Methods and approximated numerically an error is incurred. Some knowledge about the error is available in advance of the implementation of a specific FEM approximation. This type of error analysis, so called a priori, is valuable for the analysis of convergence and other proofs that may be important about an FEM scheme in general. Although useful in these applications it is rare that an a priori error estimate accurately quantifies the error present in a numerical solution. As such in a practical context the a priori error cannot be used to indicate the quality of a solution in a practical sense.

In contrast, a posteriori error analysis uses the computed numerical solution u_h itself to estimate and quantify the true the error. Where a posteriori error estimates are available and accurate this has the obvious advantage that it describes the error in the approximation without the need for an analytical solution. This replaces the heuristics and guess work that had previously guided the error analysis of the FEM approximation in settings where no analytical solution was available [1].

Since the initial pioneering methods of Babuska [3] [2] in this field many a posteriori error methods have been presented. Many of these have been shown to be effective in context. The emphasis in the literature has moved now from the discovery of new a posteriori estimates to the testing of the limitations of existing a posteriori estimates. All this is done primarily with the goal of validating the extensively used self-adapting FEM packages.

5.1 Gradient Recovery

The FEM provides an approximation to an unknown function, however the gradient of the function may also be of particular interest also. The gradient in the approximation is discontinuous over element boundaries but can be smoothed out by post-processing. In fact under certain circumstances the smoothed gradient is much closer to the gradient of the true solution [1]. This leads fairly directly to one of the simpler a posteriori error estimates which is achieved by comparison of the approximated gradient before and after this post-processing.

Consider again the steady state one dimensional Heat Equation 3.2 and here restating its weak formulation for ease.

Find $u \in H_0^1(0,1)$:

$$\int_0^L u'v'dx = \int_0^L fv\,dx \qquad \forall v \in H_0^1(0,L)$$

$\forall v \in H_0^1(0,L)$

With

$$\int_0^L u'v'dx = a(u,v)$$

It is clear that the bilinear form is symmetric i.e. a(w, v) = a(v, w) and it can be shown that it is coercive as described in the Lax Milgram theorem A. Hence, $a(w,v)$ is an inner product $(w,v)_a$ on $H_0^1(0,L)$.

We can then define the associated energy norm.

$$\|w\|_a = |(w,w)_a|^{\frac{1}{2}} \tag{5.1}$$

If we measure the true error in the energy norm we get.

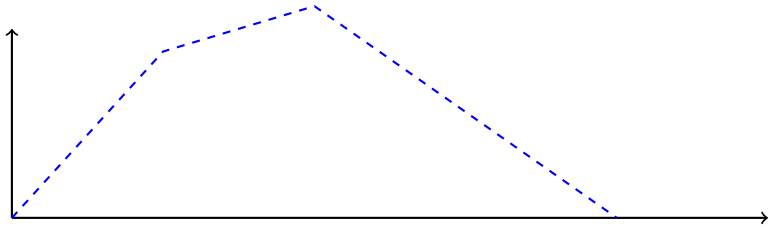
$$\|u - u_h\|_a^2 = \int_0^L u' - u_h' dx \tag{5.2}$$

Which is the L_2 norm of of the difference between the gradient and the approximated gradient which is sometimes called the H^1 semi-norm.

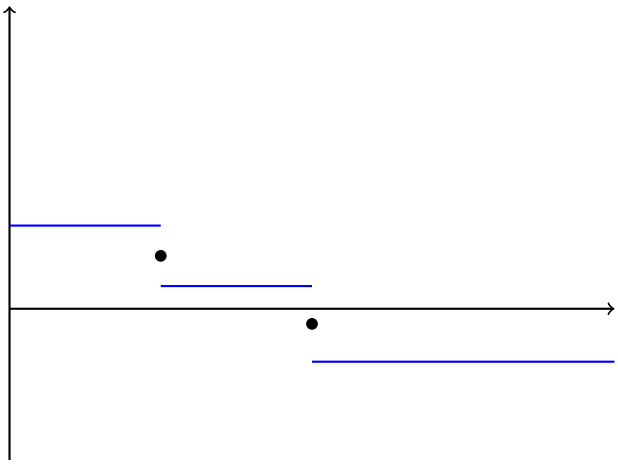
If we had the true gradient we would now be able to measure the true error in the energy norm. As that is unavailable we will use an estimate of the true gradient which will be obtained by some suitable post- processing of the FEM approximation to the solution. Let the post processed FEM approximated gradient be denoted by $G_h(u_h)$ then the error indicator for 3.2 is given by:

$$\eta^2 = \int_0^L |G_h(u_h) - u_h|^2 dx \tag{5.3}$$

Clearly there are as many different error indicators as there are ways of processing the FEM approximation to the gradient. To describe one way we consider an FEM approximation to an elliptical.



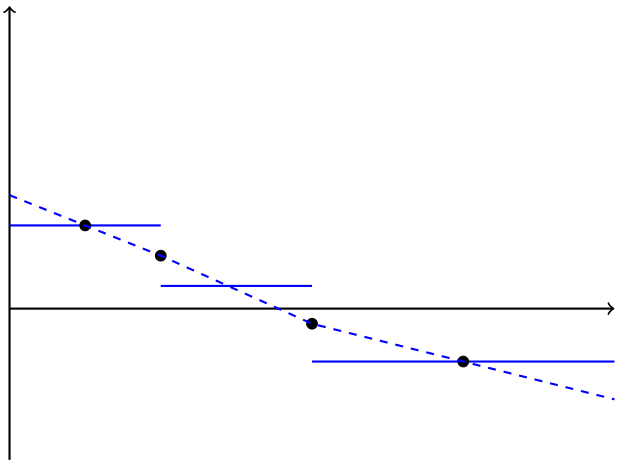
The gradient of this approximation is a discontinuous step-like function, undefined at the nodes of the space mesh.



Intuitively a better approximation would be given by a function defined in the whole domain. A simple but logical one to use could be a continuous piecewise linear function as we have the architecture in our code to linearly interpolate sets of points. We take the midpoint of the FEM approximated gradient where currently our gradient is undefined and hence where we expect post-processing to have the greatest effect.

More formally results exist to support this choice. In the case of a uniform partition the "super-convergence" of the centroid of two elements is shown by [12]. Though true "super-convergence" may not be assured on our more general partition our method nonetheless should be an improvement on the untreated gradient.

We also need to decide what to do with the elements at each side as our current method will not produce points for on the boundaries. We will choose to have the interpolant take the value of the gradient in the centre of the elements.



5.2 Effectivity Index

In fact this method gives a very good approximation to the error indicator under certain circumstances. The effectivity index is a measure of how close the error indicator comes to the true solution and for this indicator is taken to be $\frac{\eta}{\|u-u_h\|_a}$. For the elliptical PDE 3.7 we see the effectivity indexes in table 5.2 and the convergences in figure 9.

$Mesh$	DOFs	$\|u-u_h\|_{a(0,1)}$	η	Effectivity Index
1	4	0.821484	0.921108	1.121273
2	7	0.401488	0.421345	1.049459
3	13	0.199506	0.201079	1.007884
4	25	0.0995961	0.0993786	1.0021886
5	49	0.0497784	0.0496375	0.9971695
6	97	0.0248868	0.0248401	0.9981235

We have argued thus far numerically and otherwise that our indicator η is a measure of the true error in an elliptic PDE. This dissertation however concerns space adaptation in parabolic PDEs. We therefore must extend our logic slightly to include the parabolic case. It may be noted that in problem 4.1 we look for a solution of the form $u(\cdot, t)$ where the function u lives on some specific time t. So intuitively it seems that an error indicator based on the gradient may work equally well in this time dependent setting. In fact in problem 4.1 when spacial components are discretised with the FEM and give rise to the bilinear functional $a(u, v)$ which is the same as the one that arises in our elliptical problem 3.7.

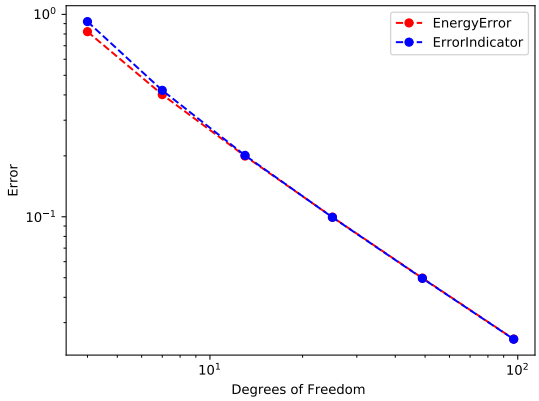


Figure 9: we can see that the η is very close to the true error even for a coarse mesh. Although we note that the error estimator is not always larger than the true error

With this justification in mind we investigate numerically the effectivity indexes in some of our parabolic model problems. Below we see the results of 4.2a.

$Mesh$	DOFs	$\|u - u_h\|_{a(0,1)}$	η	Effectivity Index
1	4	1.63002	1.71983	1.05501
2	7	0.862629	0.878275	1.018138
3	13	0.435436	0.43934	1.008966
4	25	0.218313	0.219703	1.006367
5	49	0.109433	0.109858	1.003884
6	97	0.0548128	0.0549298	1.002135

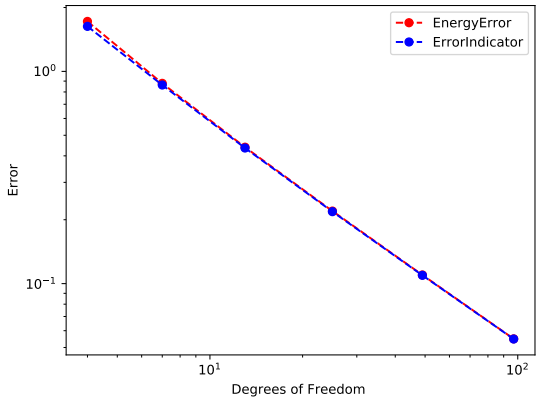


Figure 10: The error indicator convergences uniformly to the true solution and is always larger than it

We now compare the above results for 4.2a to our results for our other parabolic model problem 4.3a.

$Mesh$	DOFs	$\|u - u_h\|_{a(0,1)}$	η	Effectivity Index
1	4	0.29131	0.108467	0.372342
2	7	0.074447	0.0201663	0.270881
3	13	0.0229916	0.0135003	0.587184
4	25	0.00855775	0.00718688	0.8398095
5	49	0.00381445	0.00363495	0.9529421
6	97	0.00184453	0.00182187	0.9877150

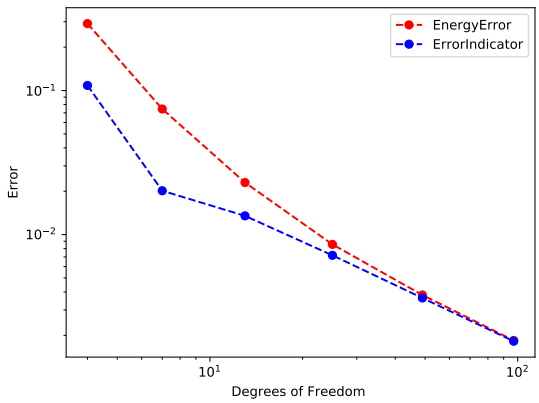


Figure 11: Here we have some interesting interaction between the η and the true error. The error indicator converges from below and sharply at first. This may present a problem for a self-adapting code specifically with regards to coarsening of a mesh. It may be necessary to add some exceptions to maintain a minimum number of DOFs obviously at least two, but possibly more where necessary

5.3 Implementation

To make the most use of our existing member functions in the SpaceMesh class we should make sure our gradient recovery vector match the space nodes. This will allow the interpolant to be completely defined by the new vector and a reference to a grid. This means that the first and last point though should be defined on the boundary. In practise this means extrapolating the line segment in the final interval on each side to the boundary and taking its value there. A function which does this is included for illustration in appendix I.

6 Design of Adaptive Algorithms

The purpose of adaptive FEM techniques can be thought of in different ways. It is not simply the case that adding nodes where there are large errors will lead to a "better" solution in any common sense way. It does however have interesting implications for the design for error analysis and the design of scientific software. Possible advantages include:

1. Equidistributed errors may serve certain applications better for example if we are studying the effects of changing parameters.

2. If adaptivity is found to be effective we may be able to achieve a comparable quality of solution with the same or fewer degrees of freedom. This would save valuable computing resources if we need to run our code multiple times or for large matrix systems.

3. The choice of mesh, initial set-up notwithstanding, is selected automatically by the algorithm. This limits the calibration needed by the end user and so allows for a professional finish on a software package for the FEM.

The rest of this chapter will describe an algorithm that can achieve the equidistribution of the errors in space. Then we will explore the implementation in principle and in practice. In subsection 6.1 we reflect on how we have been implementing the Implicit Euler thus far and explore the methodology that will allow us to both refine and coarsen our mesh between time steps. In subsection 6.2 we introduce the algorithm for space adaptation. In subsection 6.3 we look at how the decision to refine and coarsen is taken in the code.

6.1 Common Mesh Refinement

Consider the equation 4.2a and its fully discrete form defined on a suitable mesh $\Omega_h^{\Delta t}$

$$\begin{aligned} (u_h^{m+1},\, v_h) + \Delta t a(u_h^{m+1}, v_h) &= (u_h^m,\, v_h) \\ (u_h^0) &= (u_0, v_h) \quad \forall v_h \in V \qquad 0 \le m \le M-1 \end{aligned}$$

Here,

$$a(w,v) = \int_0^L w'v'dx \qquad (w,v) = \int_0^L wvdx$$

The right hand side for a non-changing mesh is fairly straightforward calculation as u_h^{m-1} and v_h are both piecewise linear and defined on the same mesh. All we need to do in this case is use quadrature with our test function over each interval or, as we have done, use the mass matrix multiplied by the previous solution Mu_h^{m-1} as explained in [7].

However, when the mesh evolves from time to time the mass matrix method is no longer meaningful and the integral $(u_h^m,\, v_h)$ becomes more difficult to evaluate. To do this we must consider the mesh where v_h is defined which is here the mesh at time $m+1$ and we must consider that the two functions u_h^m and v_h are piecewise linear and so we must be careful when using quadrature across the element boundaries on either mesh.

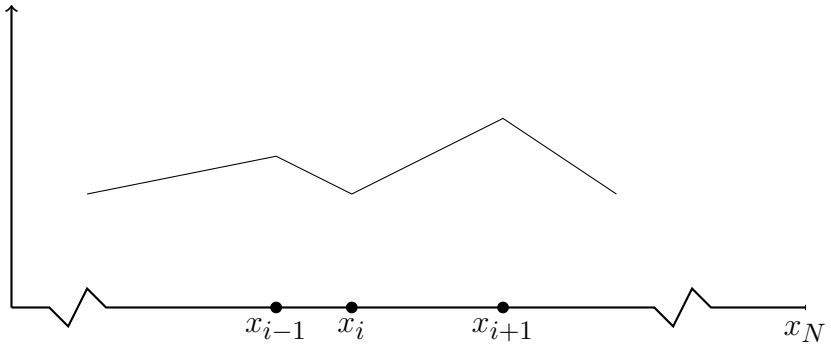


Figure 12: Approximation to u at time m

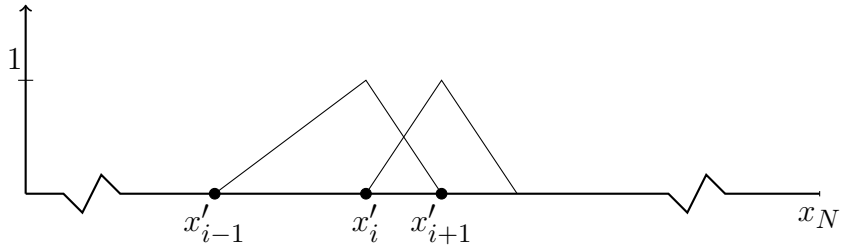


Figure 13: Basis function a time m+1

A safe way to implement $(u_h^m,\, v_h)$ is to loop the quadrature with reference to a common mesh like the one in figure 14. To accomplish this we notice that a common mesh is simply

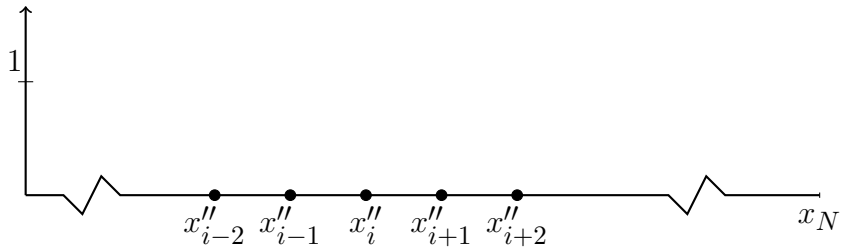


Figure 14: A common mesh

an instance of a the meshes we have previously designed and so we will need a new object from our SpaceMesh class. Then we need some new methods.

```
class SpaceMesh
{
public:
    void CommonMesh( SpaceMesh& firstmesh, SpaceMesh& secondmesh );
    void Range( double lowerlimit, double upperlimit,
    std::vector<double>& Nodes);
    double TestFunctions(int nodeIndex, double x);
    };
```

The CommonMesh method is a simple "copy erase if" algorithm made easy by the utilities in the ¡algorithm¿ library in C++11 or later. Called on our new instance of SpaceMesh it creates the common mesh required for reference.

The Range method will return all the nodes within a given range as a dynamic array. This is a design choice rather than a necessity. We choose to create an outer loop over our mesh in the current time then an inner loop which loops over the common mesh to see how many subintervals to choose. Therefore Range will be called on the common mesh object.

We can create a method TestFunctions which can be called at the current time. Then for every interval in the current mesh object we have a hat function and also our outer loop can be used as the index for said hat function.

We will also need to utilise our interpolating function with a reference to the mesh at the previous time to interpolate the previous solution.

Clearly this and all the methods thus far mentioned are straight forward to test independently.Once reliability of the code is established we we can create the method needed to build (u_h^m, v_h). This can be simply described as the test function and previous solution meeting on the common mesh and the quadrature quadrature being carried out there. Two methods which when used together can do this are given for illustration in J

```
void AdaptiveHeatEquation::BuildRHS()
{
mpRHS.clear();
std::vector<double> intervals;
        //create the common mesh from the two earlier ones
refinedsmesh.CommonMesh(mpsmesh, oldmesh);
double integral;
        //loop over internal nodes ignoring half hats for now
for (int i = 1; i<mpsmesh.meshsize(); i++)
{
    integral=0;
        //read the common mesh intervals contained
    refinedsmesh.Range(mpsmesh.ReadSpaceNode(i-1),
    mpsmesh.ReadSpaceNode(i+1), intervals);
                //loop over contained intervals
    for(int j = 0; j<intervals.size()-1; j++)
    {
        integral = integral + IntegrateBasisWithU(i, intervals.at(j),
                    intervals.at(j+1));
    }
        //save the results of integration on interval on current mesh
    mpRHS.push_back(integral);
}
}
```

6.2 Space Adaptivity

In the previous section we described an a posteriori error indicator derived from post processing the gradient of our approximation. This indicator the possibility of adapting

the space mesh of our Finite Element Method either manually or algorithmically during implementation. In fact, our specific error indicator is one of the first initially introduced for this purpose by [3] although it was derived and explained differently there [1].

The algorithm for a parabolic problem consists of:

---

for all Time steps **do**

calculate u_h

calculate the error indicator

save the elements that are too large or small

refine and coarsen

end for

An efficient way of doing this is adding some utility methods to our SpaceMesh class which allow us to insert and remove nodes from an array. We add a set of methods to our Solver class for calculating and comparing the error indicators to the tolerances. We can make our code read similar to above.

```
for(int j = 0; j<mptimemesh.NumberOfTimeSteps(); j++)
{
    BuildSystemAtTimeStep();
    SystemSolver();
    mpPreviousSolution = mpx;

    SaveIntervalsForRefinement();
    SaveIntervalsForCoarsening();
    mpsmesh.BisectIntervals(intervalsForRefinement);
    mpsmesh.CoarsenIntervals(NodesForRemoval);
}
```

6.3 Refining and coarsening

In practise there are still technical details and design decisions involved in making the above code run. These decisions are often brushed over or taken for granted in the

literature. This seemed a little odd to us as we saw many possible ways of both refining and of coarsening a mesh based on reliable a posteriori data. These different choices would lead to different outcomes and different challenges in the software. Of the two processes refinement is clearly the simpler one. Our implementation simply compares the error indicator over each element to a tolerance and if the $\eta > tolerance$ then we bisect the interval.

6.3.1 Coarsening

Coarsening is less straightforward. Most approaches to this problem will not be able to be applied on the boundary interval in the same way. We experimented with several ways of doing this

1. Take the norm of the error indicator across two elements and remove the middle node if that value is below a tolerance. This is method depends on some consistency in the error distribution at a given time step but works overall as the refinement prevents very large errors from "trapping" smaller errors. It also has the advantage that it does yield safe code without worries about the unexpected loss of nodes.

2. Take the error indicator over each element and then if below a certain tolerance remove both associated nodes and bisect the interval. This effectively redistributes the error however it needs exceptions on the boundary and can lead to some very difficult to debug code.

3. Do as above but simply remove one of the nodes i.e. always the upper or lower. This can be written in such a way as to avoid the boundaries and ugly exception lists. Also, even if the decision is sub-optimal for small enough step size Δt the refinement method should handle it quickly.

4. Once again do as above but compare the errors over the elements to the left and right and take the node separating the smaller two. This is possibly the neatest solution but still needs exceptions on the boundaries.

In fact there is no reason to restrict yourself to one or another of the methods. A useful feature of the modular code presented in 6.2 is that we can trade different coarsening

methods in and out as we like. In fact we implemented all of them but chose to use the third method for the production of our numerical experiments.

A final note on coarsening which we touched on in the caption to figure 11 is one of rapid hollowing out of a mesh. For our model problem 4.3a the error indicator is smaller than the true error. This is exacerbated by the fact that u_h has large changes in gradient in its initial conditions but then in the final time is almost a linear function. This leads to a situation where at some time step the coarsening function removes all but the boundary nodes and typically this happens instantly. When initially implemented our solver was not robust enough to handle only one element at a time step. This issue needs to be considered carefully and in advance of implementation to avoid spending large amounts of time re-writing code or finding ways around.

7 Final Results

In this section we will look at the implementation in practise. We first show an example of a self-adapting solution at various times. Then in subsection 7.2 we perform experiments varying the tolerances and observing the DOF that this produces in the final time.

7.1 Adaptation in

Below we demonstrate our adaptive solver for our model problem 4.2a and refinement tolerance of 0.8 and coarsening tolerance of 0.2. We purposely choose an sub-optimal starting mesh simply to demonstrate both the coarsening and refinement.

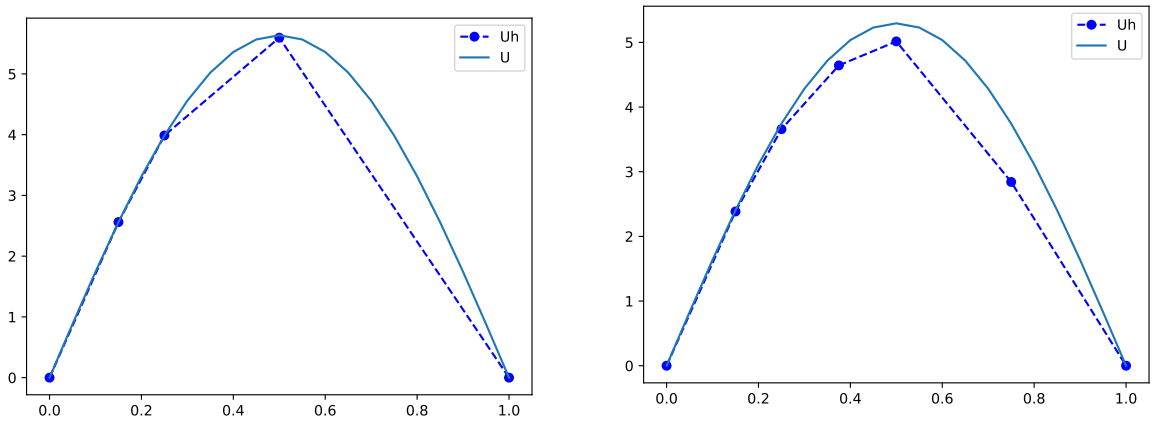


Figure 15: First and Second Time step

7.2 Testing the Adaptive Solver

Now we investigate the refinement of the solver. In the first two figure we test the refinement function isolation by setting the coarsening tolerance to zero. In the second we do the same but investigate error distributions by using the true energy error to refine and coarsen thereby creating a base of reference. Then in the last figure we look at the refinement tolerance again but in an example that has coarsening as well.

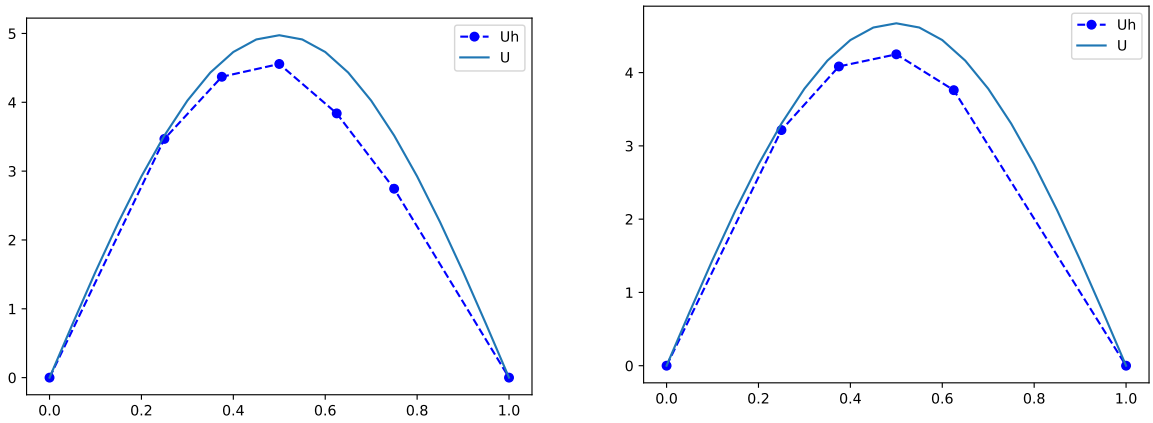


Figure 16: Third and fourth time step

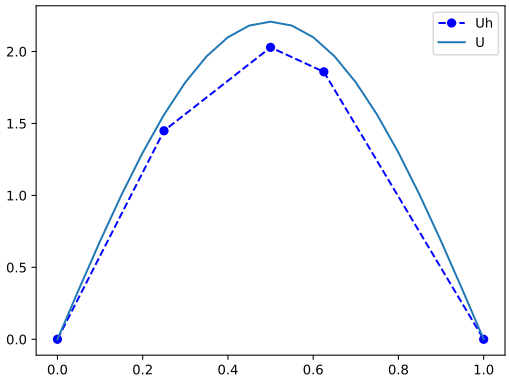


Figure 17: Final time

Iteration	Tol1	DOFs
1	1	9
2	$\frac{1}{2}$	15
3	$\frac{1}{4}$	27
4	$\frac{1}{8}$	41
5	$\frac{1}{16}$	51
6	$\frac{1}{32}$	57
7	$\frac{1}{64}$	79
8	$\frac{1}{128}$	95
9	$\frac{1}{256}$	103

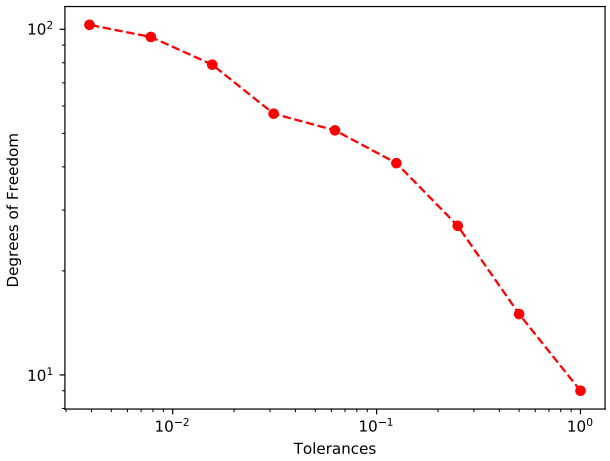


Figure 18: Here we have used our a posteriori error estimate to refine the mesh.

Iteration	Tol1	DOFs
1	1	7
2	$\frac{1}{2}$	17
3	$\frac{1}{4}$	41
4	$\frac{1}{8}$	93
5	$\frac{1}{16}$	163
6	$\frac{1}{32}$	285
7	$\frac{1}{64}$	367
8	$\frac{1}{128}$	633
9	$\frac{1}{256}$	905

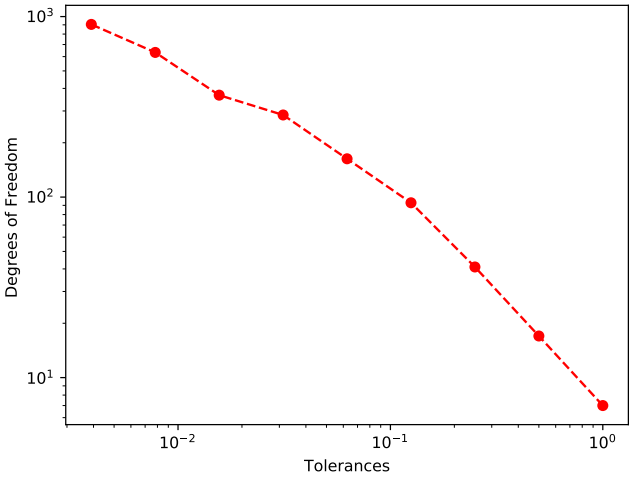


Figure 19: Here we make use of the true energy error in stead of the error indicator. We see a smoother rate of adaptivity than for the error indicator and far more DOFs used. Something which may be at the root of the disparity is the accuracy with which the error indicator distributes the error. We know from the effectivity index that our indicator converges to the true error we do not however have any guarantee that the indicator is accurate with regards to the error distribution.

Iteration	Tol1	Tol2	DOFs
1	1	$\frac{1}{2}$	9
2	$\frac{1}{2}$	$\frac{1}{4}$	9
3	$\frac{1}{4}$	$\frac{1}{8}$	19
4	$\frac{1}{8}$	$\frac{1}{16}$	26
5	$\frac{1}{16}$	$\frac{1}{32}$	35
6	$\frac{1}{32}$	$\frac{1}{64}$	45
7	$\frac{1}{64}$	$\frac{1}{128}$	53
8	$\frac{1}{128}$	$\frac{1}{256}$	62
9	$\frac{1}{256}$	$\frac{1}{512}$	65

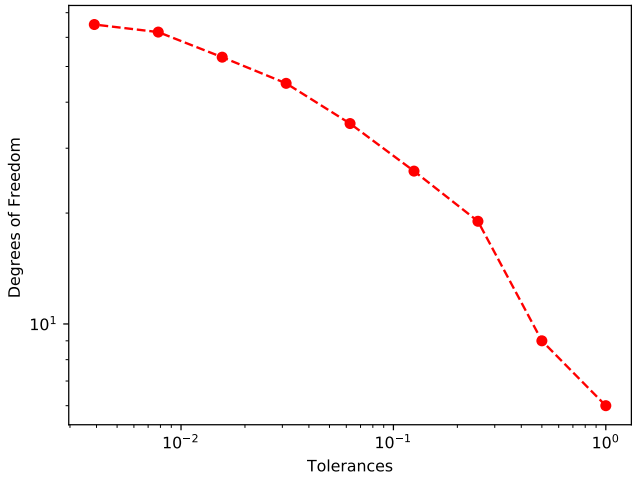


Figure 20: With comparison to the first figure which used the a posteriori error indicator we see that the coarsening tolerance is activating and yielding a smaller number of degrees of freedom in the final time. The rate of change becomes smoother for lower tolerances.

Iteration	Tol1	Tol2	DOFs
1	1	$\frac{1}{2}$	6
2	$\frac{1}{2}$	$\frac{1}{4}$	8
3	$\frac{1}{4}$	$\frac{1}{8}$	12
4	$\frac{1}{8}$	$\frac{1}{16}$	12
5	$\frac{1}{16}$	$\frac{1}{32}$	42
6	$\frac{1}{32}$	$\frac{1}{64}$	52
7	$\frac{1}{64}$	$\frac{1}{128}$	56
8	$\frac{1}{128}$	$\frac{1}{256}$	57
9	$\frac{1}{256}$	$\frac{1}{512}$	72

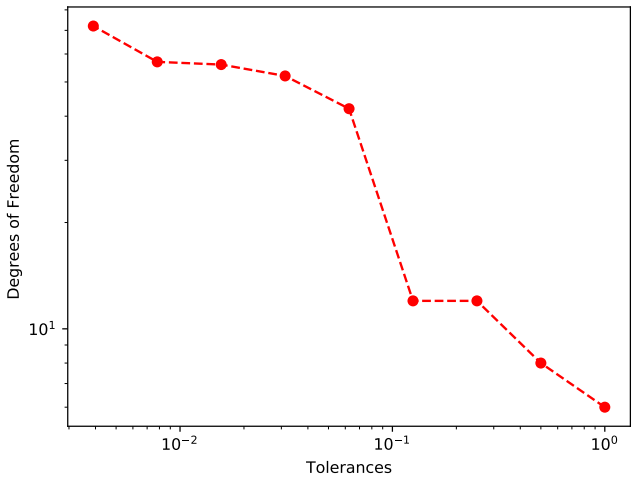


Figure 21: When we investigate the refinement and coarsening for the true energy error we find some large fluctuations. The capacity of the solution to change very suddenly in DOFs is attributable to the fact that the coarsening function can remove almost all the nodes in one step but refinement is a constrained process. An interesting extension would be to build a coarsening method which can only remove a proportion of the DOFs or a refinement method which can do more than bisect.

8 Conclusions

In conclusion, the task of building an adaptive solver is a difficult one but when approached in a modular way it can be built up slowly and in a manageable way. As stated at the outset this was an ambitious goal and so where necessary we chose the simplest examples that still embodied the principles that we were investigating. To do this we began with an elliptical PDE in subsection 3.3 and solved it on regular and irregular meshes. Then we used that base to solve the parabolic case in subsection 4.2. We chose an error indicator for space adaptivity that was fundamental enough to be explained and implemented easily but left the software infrastructure so that we could easy swap it for another indicator or the true solution as we showed in subsection 7.2. And then our final implementation challenge was designing refinement and particularly coarsening functions that could provide robust results. This was only partially successful. We also did this in a modular way to lay foundations for other methods to coarsen and refine in future studies. Object orientation has been central to this development process and the crucial classes that we have defined are:

1. A class of Tridiagonal Marices from which we derived the Stiffness and Mass matrix. As a project scaled further methods and functionality. For example in an extension to this project to create entries could be created by quadrature which could be tested separately. This more sophisticated initialisation would have access to all the utility functions that went before

2. A class of Space Meshes. This class has been used to allow concepts of FEM meshes to be implemented intuitively and code based on it to be written in a readable way.

3. A class of Time Meshes. This class was developed parallel to the Space Mesh but kept separate to avoid issues of access undermining the code.

4. A class of PDEs defined as abstract with virtual functions to manage all the possible characteristic of a PDE that we would study. The derived classes then formed our interface allowing different PDEs to be passed simply to solvers.

In this way we met our objective which was the design of safe scalable code.

Our other objective was to investigate in what sense an adaptive solver can be called "efficient".

To set the background for this we studied the convergence of the FEM. In the elliptical case we showed an order of convergence of n^{-2} and for the parabolic case with step size $\Delta t = n^{-2}$ we also found the order of convergence n^{-2}. Another important context for our concept of efficiency was the accuracy of the error estimator we used. We introduce the gradient recovery method from the creation of an indicator η in section 5. We found that it converged with the true energy error in the case of both our model problems. With this context established we designed our adaptive algorithm. As this is not a paper on complexity in the computational sense we chose a simple algorithm to refine and a few different algorithms to coarsen. These algorithms had some issues. The most significant issue was the fact that our refinement method could in one step only bisect the existing intervals which limited the next step to at most double the DOFs. In contrast our coarsening method could remove all internal nodes in one step. This issue could create large changes in the DOFs and was a limiting factor when looking for an order of convergence in 7.1.

We found that for a refinement only experiment the error indicator inserted less nodes than the true error for each given tolerance. We explain this by observing that we cannot be sure that our chose error indicator follows the error distribution accurately only that it converges to error. This certainly may be regarded as a way that the adaptive solver is not optimally efficient. An interesting extension to this study could be to implement other error indicators and explore their representation of the error distribution further.

A Lax Milgram

Theorem A.1 (Lax Milgram). *Let V be a real Hilbert Space with associated norm $\|\cdot\|$. Also let $a(\cdot,\cdot)$ be a bilinear functional on $V \times V$ with:*

(a) $a(\cdot,\cdot)$ is coercive, i.e. there exists a positive constant $c_0 \forall v \in V | a(v,v)| \geq c_0 \|v\|_V^2$

(b) $a(\cdot,\cdot)$ is continuous i.e. there exists a positive constant $c_1 \forall v, w \in V | a(v,w) \leq c_1 \|v\|_V \|w\|_V$

and for $\ell(\cdot)$ a bilinear functional also on V such that

(c) $\ell(\cdot)$ is continuous i.e. there exists a positive constant $c_2 \forall v \in V | \ell(\cdot)| \leq c_2 \|v\|_V$

Then $\exists u \in V \ : \ a(u,v) = \ell(v) \, \forall v \ \in V$ and u is unique.

B Tridiagonal Matrices

```
class TriDiagMatrix
{
public:
    void MatrixSolver( std::vector<double> f, std::vector<double> &x )

protected:
    std::vector<double> mpDiagonal;
    std::vector<double> mpLowerDiag;
    std::vector<double> mpUpperDiag;
};

    void TriDiagMatrix::MatrixSolver( std::vector<double> f,
     std::vector<double> &x )
    {
        std::vector<double> cDiagonal = mpDiagonal;
        std::vector<double> cLowerDiag = mpLowerDiag;
        std::vector<double> cUpperDiag = mpUpperDiag;
        x.assign(mpn, 0.);
         //scaling matrix entries
    for(int i=1; i<mpn; i++)
    {
        cDiagonal.at(i) = cDiagonal.at(i)-
```

```
        (cUpperDiag.at(i-1)*cLowerDiag.at(i)/cDiagonal.at(i-1));
        f.at(i) = f.at(i)-(f.at(i-1)*cLowerDiag.at(i)/cDiagonal.at(i-1));
    }

        //solving through elimination
     x.at(mpn-1)=f.at(mpn-1)/cDiagonal.at(mpn-1);
     for(int i=mpn-2; i>=0; i=i-1)
     {
         x.at(i) = (f.at(i) - cUpperDiag.at(i)*x.at(i+1))/(cDiagonal.at(i));
     }
    }
```

C Stiffness Matrix

```
class StiffnessMatrix: public TriDiagMatrix
{
public:

    void BuildStiffnessMatrix ( SpaceMesh smesh );
    void BuildGeneralStiffnessMatrix ( SpaceMesh smesh );
    void SetParameters (double k_0, double k_L, double constant);

protected:
    const double M_PI = 2*acos(0);
    double a, mpk_0, mpk_L;
};

void StiffnessMatrix::BuildGeneralStiffnessMatrix ( SpaceMesh smesh )
{
mpn = smesh.meshsize()+1;
mpDiagonal = { a*pow(smesh.ReadSpaceMesh(0), -1)+mpk_0 };
mpLowerDiag = {0};
mpUpperDiag = { -1*a*pow( smesh.ReadSpaceMesh(0), -1 ) };

for(int i=0; i<mpn-2; i++)
```

```
    {
        mpDiagonal.push_back( a*pow(smesh.ReadSpaceMesh(i), -1) +
        a*pow(smesh.ReadSpaceMesh(i+1), -1) );
        mpLowerDiag.push_back( -1*a*pow(smesh.ReadSpaceMesh(i),-1)  );
        mpUpperDiag.push_back( -1*a*pow(smesh.ReadSpaceMesh(i+1), -1) );
    }

mpDiagonal.push_back( a*pow(smesh.ReadSpaceMesh(mpn-2), -1)+mpk_L );
mpLowerDiag.push_back( -1*a*pow(smesh.ReadSpaceMesh(mpn-2),-1) );
mpUpperDiag.push_back( 0 );
}
```

D Mass Matrix

```
class MassMatrix: public TriDiagMatrix
{
public:
    void BuildMassMatrix ( SpaceMesh smesh );
    void BuildGeneralMassMatrix ( SpaceMesh smesh );
};

void MassMatrix::BuildGeneralMassMatrix ( SpaceMesh smesh )
{
mpn = smesh.meshsize()+1;
mpDiagonal = {smesh.ReadSpaceMesh(0)*pow(3,-1)};
mpLowerDiag = {0};
mpUpperDiag = {(smesh.ReadSpaceMesh(0)*pow(6,-1))};

for(int i=0; i<mpn-2; i++)
    {
        mpDiagonal.push_back( (smesh.ReadSpaceMesh(i)+
        smesh.ReadSpaceMesh(i+1))*pow(3,-1) );
        mpLowerDiag.push_back( smesh.ReadSpaceMesh(i)*pow(6,-1) );
        mpUpperDiag.push_back( smesh.ReadSpaceMesh(i+1)*pow(6,-1) );
    }
```

```
mpDiagonal.push_back( smesh.ReadSpaceMesh(mpn-2)*pow(3,-1)) ;
mpLowerDiag.push_back( smesh.ReadSpaceMesh(mpn-2)*pow(6,-1) );
mpUpperDiag.push_back( 0 );
}
```

E Space Mesh

```
class SpaceMesh
{
public:
    void Range( double lowerlimit, double upperlimit,
    std::vector<double>& Nodes);
    void CommonMesh( SpaceMesh& firstmesh, SpaceMesh& secondmesh );
    double TestFunctions(int nodeIndex, double x);


protected:
    std::vector<double> mpSpaceNodes;
};

void SpaceMesh::Range( double lowerlimit, double upperlimit,
                std::vector<double>& Nodes)
{
    Nodes.clear();
    for(auto i:mpSpaceNodes)
        if(lowerlimit<=i&&i<=upperlimit)
        {
            Nodes.push_back(i);
        }
}

void SpaceMesh::CommonMesh( SpaceMesh& firstmesh, SpaceMesh& secondmesh )
{
    mpSpaceNodes=firstmesh.mpSpaceNodes;
    mpSpaceNodes.insert( mpSpaceNodes.end(),
    secondmesh.mpSpaceNodes.begin(), secondmesh.mpSpaceNodes.end() );
```

```
    sort(mpSpaceNodes.begin(), mpSpaceNodes.end());
    mpSpaceNodes.erase( unique( mpSpaceNodes.begin(),
    mpSpaceNodes.end()), mpSpaceNodes.end() );
}

double SpaceMesh::TestFunctions( int nodeIndex, double x)
{
        if((nodeIndex==0)||(nodeIndex==meshsize()))
    {
        return 0;
    }
    else if (x<mpSpaceNodes[nodeIndex]-ReadSpaceMesh(nodeIndex-1)||
    x>mpSpaceNodes[nodeIndex]+ReadSpaceMesh(nodeIndex))
    {
        return 0;
    }
    else if (x<=mpSpaceNodes[nodeIndex])
    {
        return pow(ReadSpaceMesh(nodeIndex-1),-1)*
        (x-mpSpaceNodes.at(nodeIndex))+1;
    }
    else if (x>mpSpaceNodes[nodeIndex])
    {
        return -pow(ReadSpaceMesh(nodeIndex),-1)*
        (x-mpSpaceNodes.at(nodeIndex))+1;
    }
}
```

F Time Mesh

```
class TimeMesh
{
public:

    void CopyTimeMesh (const TimeMesh& oldTimeMesh);
    void GenerateTimeMesh ( std::vector<double> TimeNodes );
```

```
    void GenerateUniformTimeMesh ( int mTimeSteps, double TFinalTime );
    void BisectInterval (int lowerIndex, int upperIndex);
    void GloballyBisectTimeMesh ();
    void InsertTimeNode ( double ti );
    void RefreshTimeMesh();
    void PrintTimeNodes ();
    void PrintTimeMesh ();
    double ReadTimeStep (int i);
        //reads the step size between nodes
    double ReadTimeMesh (int i);
    int NumberOfTimeSteps();
protected:
std::vector<double> mpTimeNodes;
};
```

G PDE Interface

```
class APDE
{
    public:
        friend class GeneralHeat;
        virtual double ContinuousAnalyticSolution( double x, double t );
        virtual double AnalyticGradientWRTx( double x, double t );
        virtual double EllipticalRHSfunction( double x );
        virtual double FirstBoundary( double t );
        virtual double SecondBoundary( double t );
        virtual void InitialCondition ( SpaceMesh& a_mesh,
        std::vector<double>& first_U );
    protected:
        const double M_PI = 2*acos(0);
        double a, g_0, g_L, k_0, k_L;
};
```

H A Parabolic PDE

```
class PDE_Q2: public APDE
{
    public:
        friend class GeneralHeat;
        double ContinuousAnalyticSolution( double x, double t );
        double AnalyticGradientWRTx( double x, double t );
        void InitialCondition ( SpaceMesh& a_mesh,
                        std::vector<double>& first_U );
        double FirstBoundary( double t );
        double SecondBoundary( double t );
        PDE_Q2( );

    protected:

};



double PDE_Q2::ContinuousAnalyticSolution( double x, double t )
{
     return exp(-4*t)*sin(2*M_PI*x)+x;
}

double PDE_Q2::AnalyticGradientWRTx( double x, double t )
{
    return 2*M_PI*exp(-4*t)*cos(2*M_PI*x)+1;
}

void PDE_Q2::InitialCondition ( SpaceMesh& a_mesh,
                std::vector<double>& first_U )
{
    first_U.clear();
    double var;
    for (int i = 0; i<a_mesh.meshsize()+1; i++)
    {
       var = sin(2*M_PI*a_mesh.ReadSpaceNode(i))+a_mesh.ReadSpaceNode(i);
```

```
        first_U.push_back(var);
    }
}

double PDE_Q2::FirstBoundary( double t )
{
    return 0;
}

double PDE_Q2::SecondBoundary( double t )
{
    return 1;
}



PDE_Q2::PDE_Q2(  )
{
   a= pow(M_PI, -2), g_0 =0 , g_L = 1, k_0 = pow(10, 300),
    k_L = pow(10, 300);
}
```

I Gradient Recovery

```
void GeneralHeat::GradientRecoveryFunction( SpaceMesh& relevantMesh,
      std::vector<double>& gradvec, std::vector<double>& gradrecovery ){

        //(x_0, y_0) (x_1, y_1) define the line segment to be hard coded
    double x_0 = 0.5*(relevantMesh.ReadSpaceNode(1)+
                                relevantMesh.ReadSpaceNode(0));
    double y_0 = gradvec.at(0);
    double x_1 = relevantMesh.ReadSpaceNode(1);
    double y_1 = 0.5*(gradvec.at(1)+gradvec.at(0));

    gradrecovery.push_back(y_0+
        (relevantMesh.ReadSpaceNode(0)-x_0)*(y_1-y_0)/(x_1-x_0));
```

```
                //take the midpoint of the two gradients where undefined
    for(int i = 0; i<relevantMesh.meshsize()-1; i++)
    {
        gradrecovery.push_back(0.5*(gradvec.at(i)+gradvec.at(i+1)));
    }

                //the hardcoding needs to happen on both sides
    x_0 = relevantMesh.ReadSpaceNode(mpsmesh.meshsize()-1);
    y_0 = gradrecovery.back();
    x_1 = 0.5*(relevantMesh.ReadSpaceNode(relevantMesh.meshsize())
    +relevantMesh.ReadSpaceNode(relevantMesh.meshsize()-1));
    y_1 = gradvec.back();

    gradrecovery.push_back(y_0+
    (mpsmesh.ReadSpaceNode(mpsmesh.meshsize())-x_0)*(y_1-y_0)/(x_1-x_0));
}
```

J Build RHS

```
double AdaptiveHeatEquation::IntegrateBasisWithU(
        int NodeIndex, double lowerlimit, double upperlimit ){
    auto SolutionWithBasis = [&](double x)
        { return mpsmesh.TestFunctions( NodeIndex, x)*
        GeneralInterpolant(x, oldmesh, mpPreviousSolution); };
    return gauss<double, 7>::integrate(SolutionWithBasis, lowerlimit,
     upperlimit);
        }

void AdaptiveHeatEquation::BuildRHS() {
        refinedsmesh.CommonMesh(mpsmesh, oldmesh);
        double integral;
        for (int i = 1; i<mpsmesh.meshsize(); i++)
        {
        integral=0;
        refinedsmesh.Range(mpsmesh.ReadSpaceNode(i-1),
```

```
        mpsmesh.ReadSpaceNode(i+1), intervals);

    for(int j = 0; j<intervals.size()-1; j++)
    {
        integral = integral + IntegrateBasisWithU(i,
        intervals.at(j), intervals.at(j+1));
    }
    mpRHS.push_back(integral);
}
}
```

References

[1] M Ainsworth and JT Oden. A posteriori error estimation in finite element analysis (wiley-interscience, new york, 2000). *MR1885308 (2003b: 65001)*, 2000.

[2] I Babuska and A Miller. A-posteriori error estimates and adaptive techniques for the finite element method. Technical report, MARYLAND UNIV COLLEGE PARK INST FOR PHYSICAL SCIENCE AND TECHNOLOGY, 1981.

[3] Ivo Babuska and Werner C Rheinboldt. A-posteriori error estimates for the finite element method. *International Journal for Numerical Methods in Engineering*, 12(10):1597–1615, 1978.

[4] R. Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bull. Amer. Math. Soc.*, 49(1):1–23, 01 1943.

[5] Kenneth Eriksson and Claes Johnson. Adaptive finite element methods for parabolic problems i: A linear model problem. *SIAM Journal on Numerical Analysis*, 28(1):43–77, 1991.

[6] Claes Johnson. Error estimates and adaptive time-step control for a class of one-step methods for stiff ordinary differential equations. *SIAM Journal on Numerical Analysis*, 25(4):908–926, 1988.

[7] Mats G Larson and Fredrik Bengzon. *The finite element method: theory, implementation, and applications*, volume 10. Springer Science & Business Media, 2013.

[8] Pedro Morin, Kunibert G Siebert, and Andreas Veeser. A basic convergence result for conforming adaptive finite elements. *Mathematical Models and Methods in Applied Sciences*, 18(05):707–737, 2008.

[9] Joe Pitt-Francis and Jonathan Whiteley. *Guide to scientific computing in C++*. Springer, 2012.

[10] Kunibert G Siebert. A convergence proof for adaptive finite elements without lower bound. *IMA journal of numerical analysis*, 31(3):947–970, 2011.

[11] Jürgen Topper. Option pricing with finite elements. *Wilmott Magazine*, pages 84–90, 2005.

[12] Miloš Zlámal. Superconvergence and reduced integration in the finite element method. *Mathematics of computation*, 32(143):663–685, 1978.