Finite Elemental Method

Mikkel Koefoed Lindtner - s205421

January 9, 2026

Contents

1 Exercise 1.1 2

2 Exercise 1.2 10

3 Exercise 1.5 20

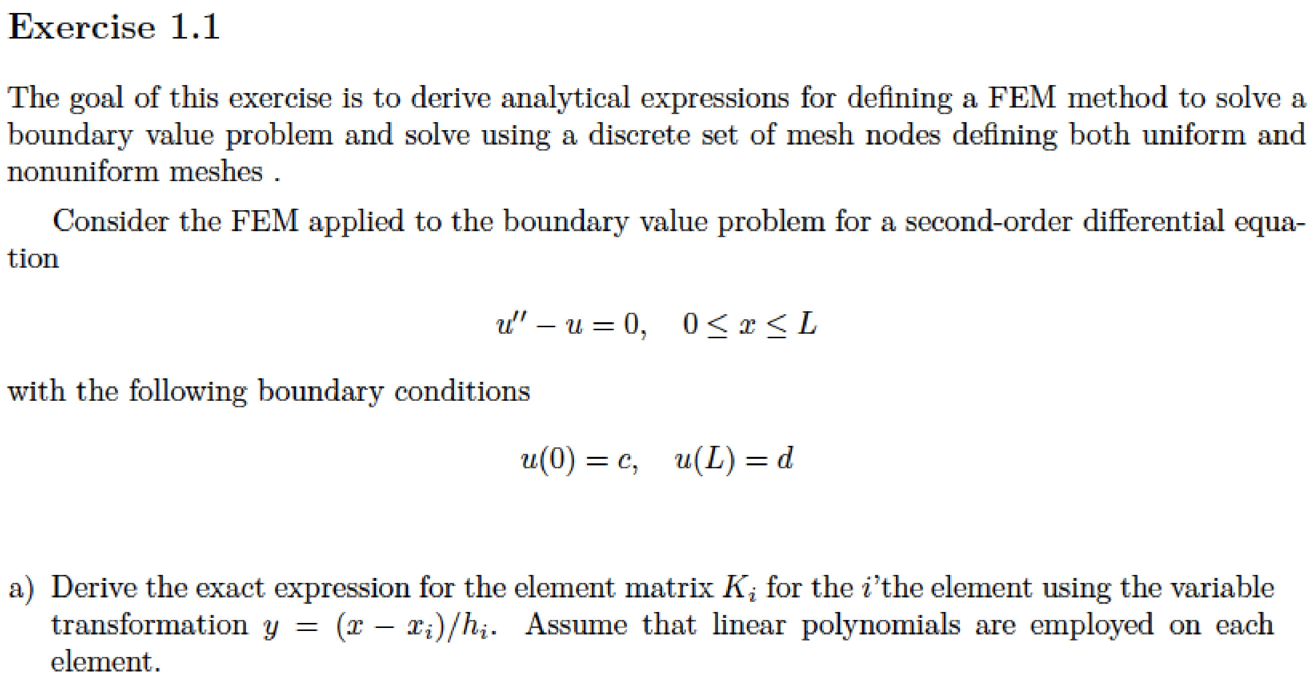
4 Exercise 1.6 31

5 Exercise 1.7 40

6 Appendix - Summary of Ai usage 56

1 Exercise 1.1

Generative Ai, Gemini has been used. The chatbot was asked to make a summary of the conversation and to make the summary academically relevant. The chatbots output was posted in google docs copy pasted into latex under "Appendix - Summary of Ai Usage". If this is not satisfactory, feel free to ask me to elaborate on any of the results.



Exercise 1.1

The goal of this exercise is to derive analytical expressions for defining a FEM method to solve a boundary value problem and solve using a discrete set of mesh nodes defining both uniform and nonuniform meshes .

Consider the FEM applied to the boundary value problem for a second-order differential equation

$$u'' - u = 0, \quad 0 \leq x \leq L$$

with the following boundary conditions

$$u(0) = c, \quad u(L) = d$$

a) Derive the exact expression for the element matrix K_i for the i'the element using the variable transformation $y = (x - x_i)/h_i$. Assume that linear polynomials are employed on each element.

From 1.6 the basis functions are defined as

$$\begin{aligned} N_1^{(i)}(x) &= N_i(x) = 1 - \frac{x - x_i}{h_i}, \quad x_i \leq x \leq x_{i+1} \\ N_2^{(i)}(x) &= N_{i+1}(x) = \frac{x - x_i}{h_i}, \quad x_i \leq x \leq x_{i+1} \end{aligned} \tag{1.6}$$

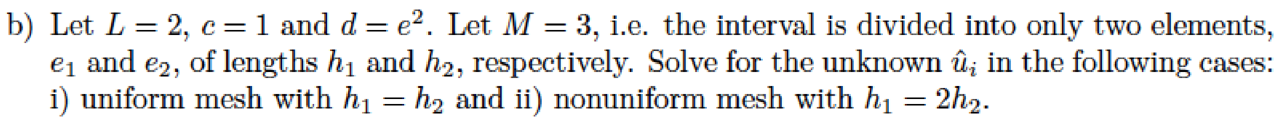
From 1.26 we know that

$$k_{r,s}^{(i)} = \int_{x_i}^{x_{i+1}} [(N_r^{(i)})'(N_s^{(i)})' + N_r^{(i)} N_s^{(i)}]\, dx\,, \qquad r, s = 1, 2 \tag{1.26}$$

Now let $y = \frac{x - x_i}{h_i}$ as described, then $dy = \frac{1}{h_i} dx$. For the limits of 1.26 we substitute $x = x_i \Rightarrow \frac{x_i - x_i}{h_i} = 0$ and $x = x_{i+1} \Rightarrow \frac{x_{i+1} - x_i}{h_i}$. By 1.1 $h_i = x_{i+1} - x_i$, so $\frac{x_{i+1} - x_i}{h_i} = \frac{h_i}{h_i} = 1$. Substituting into 1.26 we have

$$
\begin{aligned}
&h\int_0^1 \frac{1}{h_r}\frac{1}{h_s} + [1-\frac{x-x_r}{h_r}][1-\frac{x-x_s}{h_s}]\ dy \\
&= h\int_0^1 \frac{1}{h^2} + (1-y)(1-y)\ dy
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{1}{h}[y]_0^1 + h[-\frac{1}{3}(1-y)^3]_0^1 \\
&= \frac{1}{h} + \frac{h}{3}
\end{aligned}
$$

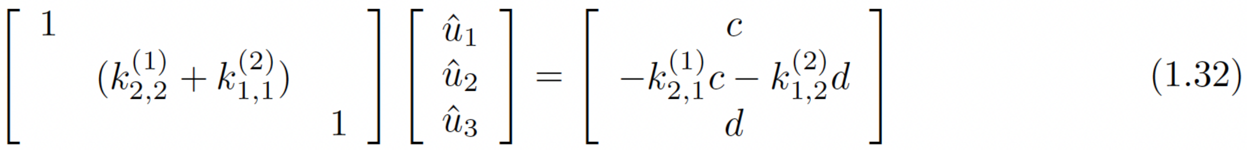


b) Let $L = 2$, $c = 1$ and $d = e^2$. Let $M = 3$, i.e. the interval is divided into only two elements, e_1 and e_2, of lengths h_1 and h_2, respectively. Solve for the unknown $\hat{u}_i$ in the following cases: i) uniform mesh with $h_1 = h_2$ and ii) nonuniform mesh with $h_1 = 2h_2$.

Inserting the requirements we have

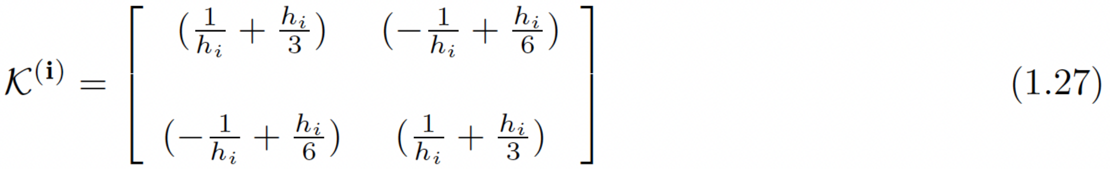
$$\begin{aligned} &u'' - u = 0, \quad 0 \le x \le 2 \\ &u(0) = 1, \quad u(2) = e^2 \end{aligned}$$

We utilize equation 1.32



$$\begin{bmatrix} 1 & & \\ & (k_{2,2}^{(1)} + k_{1,1}^{(2)}) & \\ & & 1 \end{bmatrix} \begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \end{bmatrix} = \begin{bmatrix} c \\ -k_{2,1}^{(1)} c - k_{1,2}^{(2)} d \\ d \end{bmatrix} \tag{1.32}$$

(i) We can use equation 1.27 to find the relevant k expressions



$$\mathcal{K}^{(\mathbf{i})} = \begin{bmatrix} (\frac{1}{h_i} + \frac{h_i}{3}) & (-\frac{1}{h_i} + \frac{h_i}{6}) \\ \\ (-\frac{1}{h_i} + \frac{h_i}{6}) & (\frac{1}{h_i} + \frac{h_i}{3}) \end{bmatrix} \tag{1.27}$$

assuming uniform mesh, and $h_1 = h_2$ We can find the $K^{(i)}$ matrix

$$\begin{aligned} k_{2,2}^{(1)} &= \frac{1}{h_1} + \frac{h_1}{3} = k_{1,1}^{(2)} \\ k_{2,1}^{(1)} &= -\frac{1}{h_1} + \frac{h_1}{6} = k_{1,2}^{(2)} \end{aligned}$$

Substituting into 1.32 we have

$$\begin{aligned} \begin{bmatrix} 1 & & \\ & 2(\frac{1}{h_1} + \frac{h_1}{3}) & \\ & & 1 \end{bmatrix} \begin{bmatrix} \hat{u_1} \\ \hat{u_2} \\ \hat{u_3} \end{bmatrix} &= \begin{bmatrix} 1 \\ -(-\frac{1}{h_1} + \frac{h_1}{6}) \cdot 1 - (-\frac{1}{h_1} + \frac{h_1}{6})e^2 \\ e^2 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ -\frac{h_2}{3} + \frac{1}{2h_2} - \frac{h_2}{6} + \frac{1}{h_2} \\ e^2 \end{bmatrix} \end{aligned}$$

Since $h_1 + h_2 = 2$ and $h_1 = h_2$ then $h_1 = h_2 = 1$. Solving this in maple yields gives us $\begin{bmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \hat{u}_3 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{5}{16} + \frac{5e^2}{16} \\ e^2 \end{bmatrix} \Rightarrow \hat{u}_2 = \frac{5}{16} + \frac{5e^2}{16}$

(ii)
We take care to find $K^{(i)}$ matrix using $h_1 = 2h_2$, given $h_1 + h_2 = 3 \Rightarrow h_1 = \frac{4}{3}, \quad h_2 = \frac{2}{3}$

$$\begin{aligned}
k^{(1)}_{2,2} &= \frac{1}{2h_2} + \frac{2h_2}{3} \\
k^{(2)}_{1,1} &= \frac{1}{h_2} + \frac{h_2}{3} \\
k^{(1)}_{2,1} &= -\frac{1}{2h_2} + \frac{2h_2}{6} \\
k^{(2)}_{1,2} &= -\frac{1}{h_2} + \frac{h_2}{6}
\end{aligned}$$

Subtituting into 1.32 we have

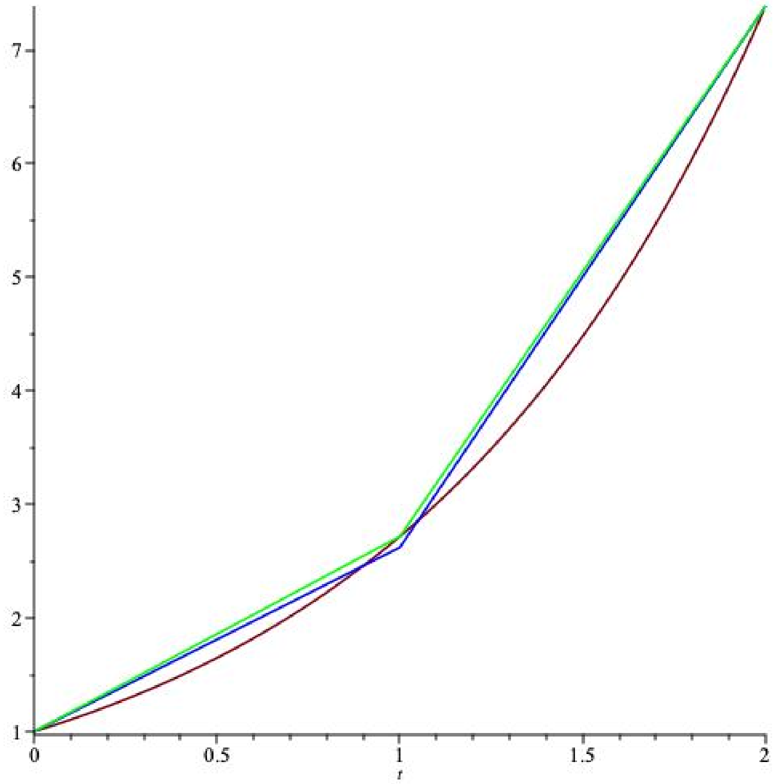
$$\begin{bmatrix} 1 & & \\ & (\frac{1}{2h_2} + \frac{2h_2}{3}) + \frac{1}{h_2} + \frac{h_2}{3} & \\ & & 1 \end{bmatrix} \begin{bmatrix} \hat{u_1} \\ \hat{u_2} \\ \hat{u_3} \end{bmatrix} = \begin{bmatrix} 1 \\ -(-\frac{1}{2h_2} + \frac{2h_2}{6}) \cdot 1 - (-\frac{1}{h_2} + \frac{h_2}{6}) \\ e^2 \end{bmatrix}$$

solving this in maple yields gives us

$$\begin{bmatrix} \hat{u}_1(x) \\ \hat{u}_2(x) \\ \hat{u}_3(x) \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{23}{105} + \frac{10}{21} e^2 \\ e^2 \end{bmatrix}$$

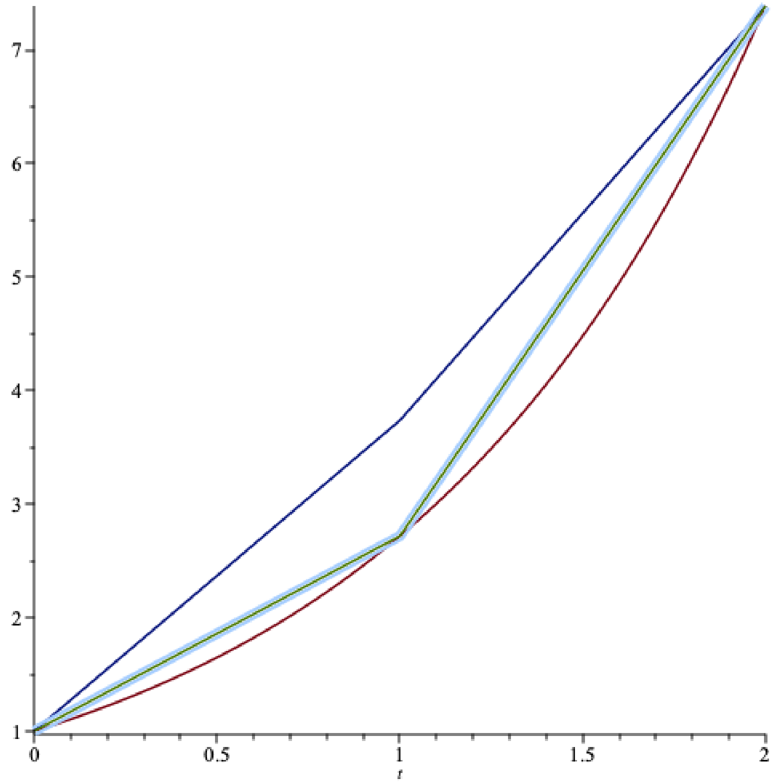
c) Consider the case with a uniform mesh with mesh sizes $h_1 = h_2$. The solution to the FEM equations determines a continuous, piecewise linear function $\hat{u}(x)$ that approximates the exact solution $u(x)$ to the problem. Plot $\hat{u}(x)$ and $u(x)$. In the same figure, include a plot of the interpolating function $u_I(x)$ that interpolates $u(x)$ at the nodes x_i, $i = 1, 2, ..., M$.

We assume that this is referencing exercise 1.1.b, to be specific we write out each u_i we find previously. It can be shown that the solution analytical solution is e^t. Letting the red graph below be the analytical solution, $\hat{u}(x)$ the blue approximation and $u(x_i)$ be the interpolation function.



d) Repeat c) for the nonuniform case ii) in b).

For the second we use the same colouring scheme, red is the interpolation, blue is the $\hat{u}$ approximation and $u_I(x)$ is the green colour



e) **Explain the difference between $u_I(x)$ and $\hat{u}(x)$. (HINT: How do we represent and define the two functions on the discrete domain? What distinguishes these definitions from each other?)**

$\hat{u}(x)$ is the approximate solution found using the the finite elemental method. By 1.2 its definition is $\hat{u}(x) = \sum_{i=1}^{M} \hat{u}_i N_i(x)$, hence we do not need to know $u(x)$ it is sufficient to know an approximation $\hat{u}(x)$ for each $i \in I$, where I is the boundary.

$u_I(x)$ is the polynomial approximation, it is defined as $u_I(x) = \sum_{i=1}^{M} u(x_i) N_i(x)$, so we must know $u(x_i)$ fo each point.

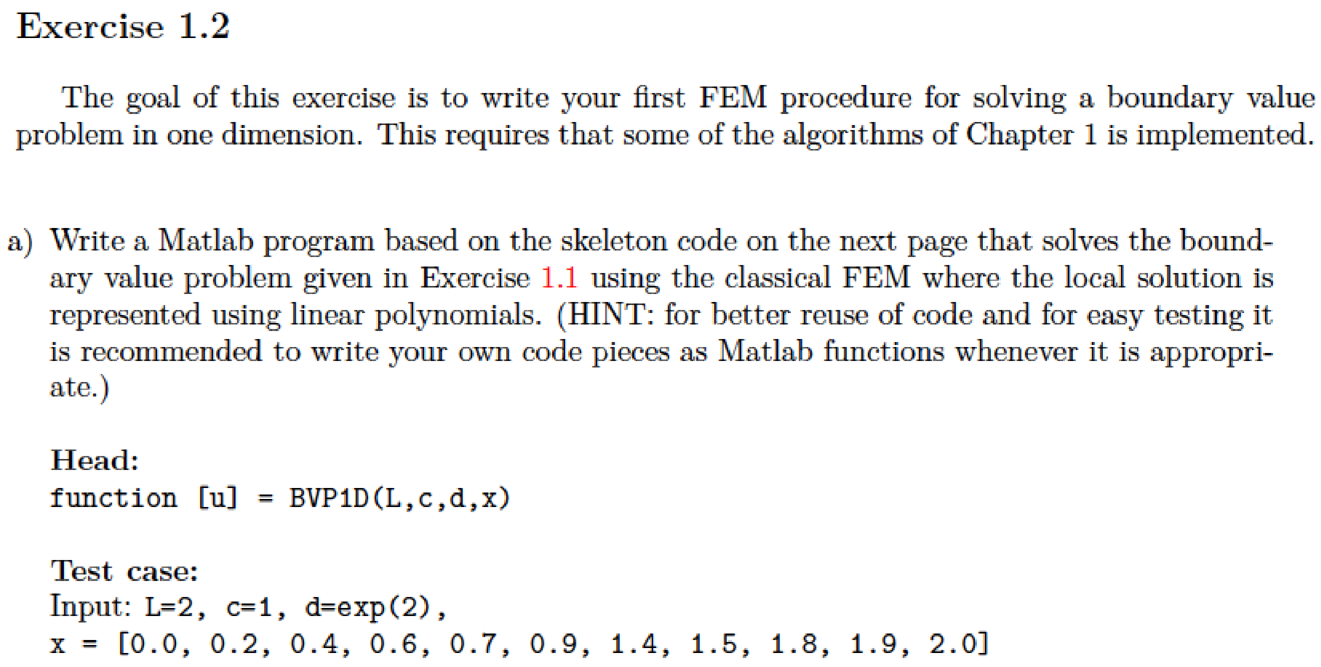
Their big difference is that we use a function $\hat{u}_i$ to approximate a value at a point x whereas for $\hat{u}(x)$ and for $u_I(x_i)$ we must know the actual function value at point x_i. This makes $u_I(x_i)$ a much harder mathematical requirement than $\hat{u}_i$.

f) **Describe the basic steps that are needed to generate a set of FEM equations to the boundary value problem.**

We want to find $A\hat{u} = b$

- Write up a weak formulation
- Calculate the appropriate $\kappa^{(i)}$ matrix
- Assemble the A matrix and b-vector from eq1.30
- Insert the boundary conditions
- Solve the system of equations for each $\hat{u}_i$
- (optional) plot it and flex on your enemies

2 Exercise 1.2



Exercise 1.2

The goal of this exercise is to write your first FEM procedure for solving a boundary value problem in one dimension. This requires that some of the algorithms of Chapter 1 is implemented.

a) Write a Matlab program based on the skeleton code on the next page that solves the boundary value problem given in Exercise 1.1 using the classical FEM where the local solution is represented using linear polynomials. (HINT: for better reuse of code and for easy testing it is recommended to write your own code pieces as Matlab functions whenever it is appropriate.)

Head:
`function [u] = BVP1D(L,c,d,x)`

Test case:
Input: `L=2, c=1, d=exp(2),`
`x = [0.0, 0.2, 0.4, 0.6, 0.7, 0.9, 1.4, 1.5, 1.8, 1.9, 2.0]`

Here is the code in julia

```julia
#Author: Mikkel Koefoed Lindtner
using LinearAlgebra
using Plots


# INPUT PARAMETERS
# L : Domain length
# c : Left boundary condition
# d : Right boundary condition
# x : 1D mesh vector x(1:{M})
L = 2
c = 1
d = exp(2)
x = [0.0, 0.2, 0.4, 0.6, 0.7, 0.9, 1.4, 1.5, 1.8, 1.9, 2.0]


function BVP1D_a(L,c,d,x)
    #Hardcoded K-values for differential equation u'' -u = 0
    M = length(x)
    h = diff(x)

    #Global Assembly
        #Assemble A (upper triangle)
        #Assemble b. (Algorithm 1)
    A = zeros(M,M)
    b = zeros(M,1)

    for i = 1:1:M-1
        k1_1 = 1/h[i] + h[i]/3
        k1_2 = -1/h[i] + h[i]/6
        k2_2 = 1/h[i] + h[i]/3

        A[i,i] = A[i,i] + k1_1
        A[i,i+1] = k1_2
        A[i+1,i+1] = k2_2

        #Use full matrix alternative
        #k2_1 = k1_2
        #A[i+1,i] = k2_1
    end
    #A[M, M-1] = 0 #I dont understand this?

    # IMPOSE BOUNDARY CONDITIONS
    # (Algorithm 2)
    #<INSERT YOUR CODE HERE>
    b[1] = c
    b[2] = b[2]-A[1,2]*c
    A[1,1] = 1
    A[1,2] = 0
    b[M] = d
    b[M-1] = b[M-1] - A[M-1,M]*d
    A[M,M] = 1
    A[M-1,M] = 0


    #visually checking A matrix is correct
    open("w1/matrix_A.txt", "w") do io
        Base.print_matrix(io, A)
    end


    # SOLVE SYSTEM
    # Solve using the Cholesky factorization of A to solve A*u=b
    # u = A \ b
    chol = cholesky(Symmetric(A, :U), check=false)
    if issuccess(chol)
        u = chol \ b
    else
        println("A is not positive definite")
        return
    end

    # open("w1/matrix_A.txt", "w") do io
    #     Base.print_matrix(io, A)
    # end

    # %% OUTPUT
    p = plot(x, u, label="FEM Solution", title="BVP 1D Solution", xlabel="x", ylabel="u")
    plot!(p, x, exp.(x), label="exp(x)", linestyle=:dash)
    savefig(p, "w1/w1_2_a.png")
```

```
end


BVP1D_a(L, c,d,x)
```

b) Make appropriate changes to the function `BVP1D` so that the number of nodes M can replace the array `x` as input in the argument list. If M is used in the call, then `x` should be computed such that the nodes are equidistant within the script. Note that the uniformity of the elements makes it possible to simplify the assembly process.

Head:
`function [u,x] = BVP1D(L,c,d,M)`

Test case
Input: `L=2, c=1, d=exp(2), M=11.`

The julia code is changed as requested

```
function BVP1D_b(L,c,d,M)
    x = range(0,L, length=M) #equidistant
    h = diff(x)
    println(x)
    println(M)
    println(h, "legnth: $(length(h))")
    #Global Assembly
        #Assemble A (upper triangle)
        #Assemble b. (Algorithm 1)
    A = zeros(M,M)
    b = zeros(M,1)

    for i = 1:1:M-1
        k1_1 = 1/h[i] + h[i]/3
        k1_2 = -1/h[i] + h[i]/6
        k2_2 = 1/h[i] + h[i]/3

        A[i,i] = A[i,i] + k1_1
        A[i,i+1] = k1_2
        A[i+1,i+1] = k2_2

        #Use full matrix alternative
        #k2_1 = k1_2
        #A[i+1,i] = k2_1
    end
    #A[M, M-1] = 0 #I dont understand this?

    # IMPOSE BOUNDARY CONDITIONS
    # (Algorithm 2)
    #<INSERT YOUR CODE HERE>
    b[1] = c
    b[2] = b[2]-A[1,2]*c
    A[1,1] = 1
    A[1,2] = 0
    b[M] = d
    b[M-1] = b[M-1] - A[M-1,M]*d
    A[M,M] = 1
    A[M-1,M] = 0
    #visually checking A matrix is correct
    open("w1/matrix_A.txt", "w") do io
        Base.print_matrix(io, A)
    end


    # SOLVE SYSTEM
    # Solve using the Cholesky factorization of A to solve A*u=b
    # u = A \ b
    chol = cholesky(Symmetric(A, :U), check=false)
    if issuccess(chol)
        u = chol \ b
    else
        println("A is not positive definite")
        return
    end

    # open("w1/matrix_A.txt", "w") do io
    #     Base.print_matrix(io, A)
    # end

    # %% OUTPUT
    p = plot(x, u, label="FEM Solution", title="BVP 1D Solution", xlabel="x", ylabel="u")
    plot!(p, x, exp.(x), label="exp(x)", linestyle=:dash)
    savefig(p, "w1/w1_2_b.png")
end


M = 11
BVP1D_b(L,c,d,M)
```

c) Validate your program by comparing with solutions of Exercise 1.1.

Looking at the produced graphs for exercise 1.2.a and 1.2.b we see that they fit well with the graphs displayed in 1.1.c and 1.1.d respectively.

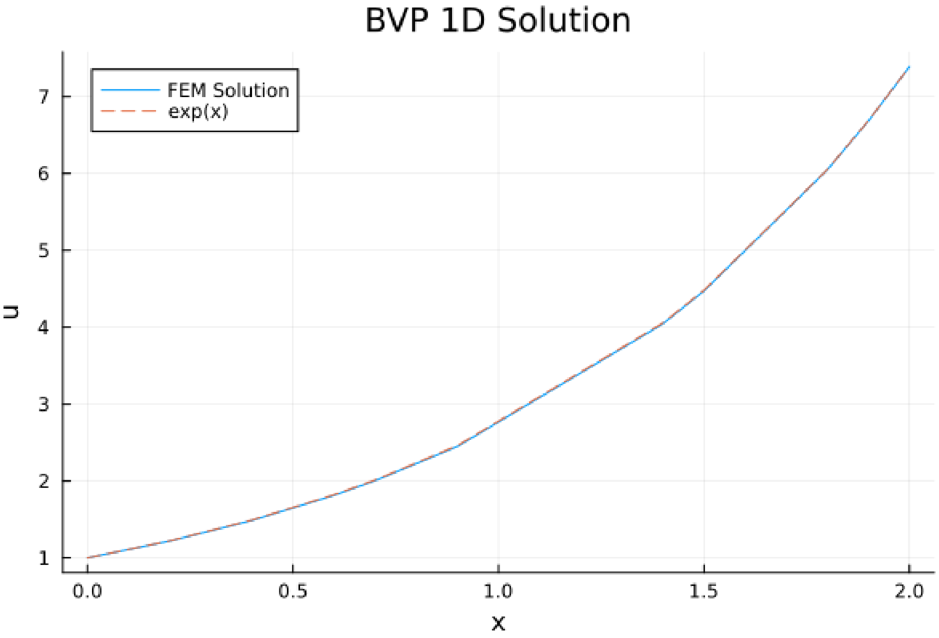
To check convergence we look at the slope between the first two points and the last two points. Note that we use a log value, so a slope of -2 will imply convergence. For the first two we have

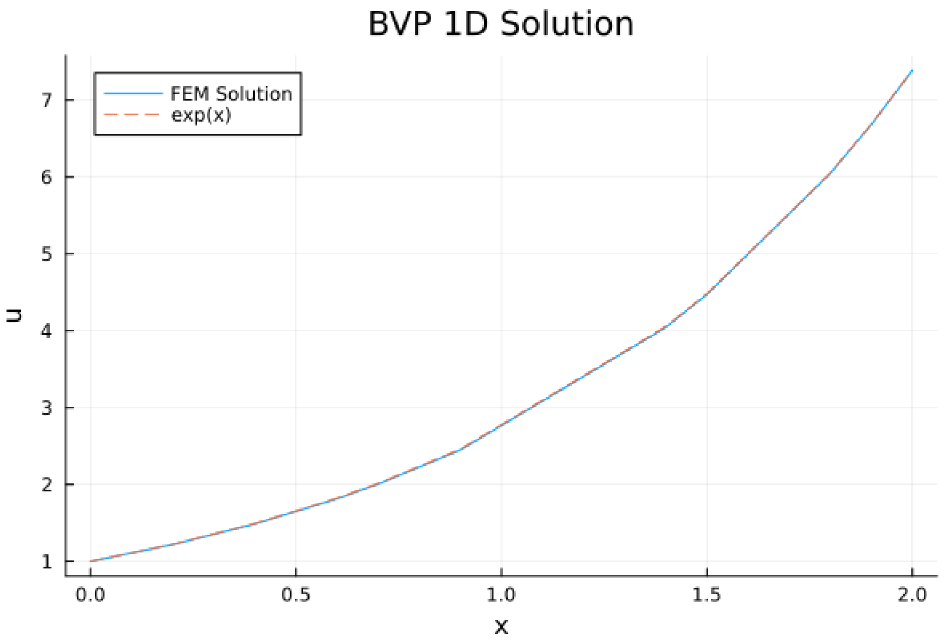
$$a_1 - a_2 = -2.935553401890821$$

And the last two points

$$a_m - a_{m-1} = -2.0149713756375967$$

The first two points are "super convergent", at least that's what the TA told me, I have no idea what that means. Ok buddy TA.





d) Compare the results computed using the function with (1.33). What is the convergence rate of the solution when the mesh size decreases, i.e. $h \to 0$? (Hint: determine an estimate for p valid in the asymptotic limit $h \to 0$ assuming that the convergence rate is $\mathcal{O}(h^p)$.)

Reviewing eq1.33

Let $u(x)$ be the solution of problem (1.14), (1.15), and let $\hat{u}(x)$ be given by

$$\hat{u}(x) = \sum_{i=1}^{M} \hat{u}_i N_i(x)$$

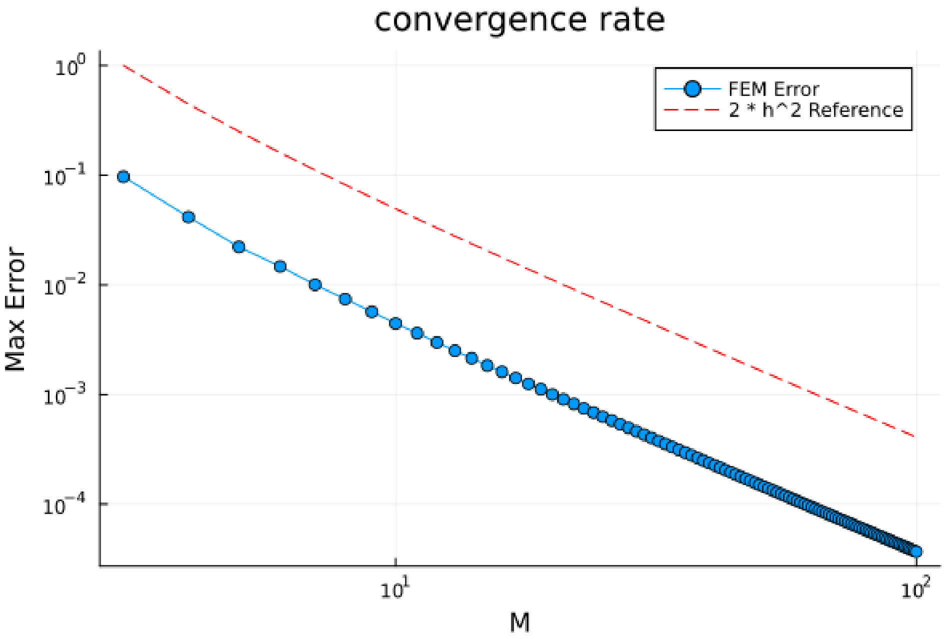
where the coefficients $\hat{u}_1$, $\hat{u}_1$,...,$\hat{u}_M$ are determined by (1.30). It can be shown that there exists a constant C such that

$$\max_{0 \le x \le L} |\hat{u}(x) - u(x)| \le Ch^2 \tag{1.33}$$

We can write the following code

```
#Exercise 1.2.D
function assembly(M,h)
    A = zeros(M,M)
    b = zeros(M,1)
    for i = 1:1:M-1
        k1_1 = 1/h[i] + h[i]/3
        k1_2 = -1/h[i] + h[i]/6
        k2_2 = 1/h[i] + h[i]/3

        A[i,i] = A[i,i] + k1_1
        A[i,i+1] = k1_2
        A[i+1,i+1] = k2_2
    end
    return A,b
```

```
end


function boundary_conditions(A,b,M)
    b[1] = c
    b[2] = b[2]-A[1,2]*c
    A[1,1] = 1
    A[1,2] = 0
    b[M] = d
    b[M-1] = b[M-1] - A[M-1,M]*d
    A[M,M] = 1
    A[M-1,M] = 0
    return A,b
end

function conv_rate(iters, L)
    M_values = Float64[]
    errors = Float64[]
    h_sq_values = Float64[]


    for M in range(3,iters)
        x = range(0,L, length=M)
        h = diff(x)

        A,b = assembly(M,h)
        A,b = boundary_conditions(A,b,M)

        chol = cholesky(Symmetric(A, :U), check=false)
        if issuccess(chol)
            uhat = chol \ b
        else
            println("A is not positive definite")
            return
        end

        error_term = maximum(abs.(exp.(x) - uhat))
        push!(M_values, M)
        push!(errors, error_term)
        push!(h_sq_values, h[1] * h[1])
    end

    p = plot(M_values, errors,
             xaxis=:log, yaxis=:log,
             marker=:circle,
             title="convergence rate",
             xlabel="M",
             ylabel="Max Error",
             label="FEM Error")

    plot!(p, M_values, 1 .* h_sq_values,
        linestyle=:dash,
        label="2 * h^2 Reference",
        color=:red)

    savefig(p, "w1/1_2_d_convergence_plot.png")
    display(p)

    #Get the slop between the first two points
    s_start = (log(errors[2]) - log(errors[1])) / (log(M_values[2]) - log(M_values[1]))
    #Get the slop between last two points (we expect approx -2.0)
    s_end   = (log(errors[end]) - log(errors[end-1])) / (log(M_values[end]) - log(M_values[en

    println("Convergence Rate (First 2 points): ", s_start)
    println("Convergence Rate (Last 2 points):  ", s_end)
end


conv_rate(100, L)
```



The plot above is produced from the code shown, the plot looks acceptable because we have found a C s.t. $\Delta err < ch^2$.

3 Exercise 1.5

Exercise 1.5

The goal of this exercise is go through the fundamental steps of formulating and implementing a Finite Element Method for a linear advection-diffusion equation. We consider the following model boundary value problem for the linear advection-diffusion equation

$$\begin{aligned} -(\epsilon u')' + (\Psi u)' &= f, \qquad x \in]0,1[\\ u(0) = u(1) &= 0. \end{aligned} \tag{1.34}$$

For simplicity we assume that the coefficients $\epsilon > 0$ and Ψ are real-valued constants on $]0,1[$.

a) Assume that u is a twice continuously differentiable function solving (1.34). Verify that u also solves the variational problem

$$a(u,v) = \ell(v), \tag{1.35}$$

for every continuous piecewise-smooth function v, such that $v(0) = v(1) = 0$, where $a(u,v) = \int_0^1 \epsilon u'v'\mathrm{d}x - \int_0^1 \Psi u v' \mathrm{d}x$ and $\ell(v) = \int_0^1 f v \mathrm{d}x$.

We first look to write up the weak definition

$$\begin{aligned} &-(\epsilon u')' + (\Psi u)' = f \\ &(-(\epsilon u')' + (\Psi u)')v = fv \\ &\int_0^1 (-(\epsilon u')' + (\Psi u)')v \; dx = \int_0^1 fv \; dx \end{aligned}$$

Splitting up the integral and focusing just on the right hand side

$$-\int_0^1 (\epsilon u')'v \; dx + \int_0^1 (\Psi u)'v \; dx = -\epsilon \int_0^1 u''v \; dx + \Psi \int_0^1 u'v \; dx$$

Recall the integration by parts

$$\begin{aligned} &(pq)' = p'q + pq' \\ &\int_0^L pq' \; dx = [pq]_0^L - \int_0^L p'q \; dx \end{aligned}$$

Focusing on the first integral, let $q = u' \Rightarrow q' = u''$ and $p = v$

$$\begin{aligned} &\int_0^1 vu'' \; dx = [qp]_0^1 - \int_0^1 p'q \; dx \\ &= [u'v]_0^1 - \int_0^1 v'u' \; dx \\ &= -\int_0^1 v'u' \end{aligned}$$

Where $[u'v]_0^1 = 0$ because of the boundary conditions. Now Taking the second integral and using integration by parts, where we let $p = v$ and $q = u$ then

$$\int_0^1 u'v\ dx = [pq]_0^1 - \int_0^1 p'q\ dx$$
$$= -\int_0^1 uv'\ dx$$

Now substituting in these results we have

$$-\epsilon\int_0^1 u''v\ dx + \Psi\int_0^1 u'v\ dx = -(-\epsilon\int_0^1 v'u'\ dx) - \Psi\int_0^1 uv'\ dx$$
$$= \int_0^1 \epsilon\ v'u'\ dx - \int_0^1 \Psi uv'\ dx$$

Since the right hand side is unchanged, this is what we wanted to show, i.e. we have

$$\int_0^1 \epsilon\ v'u'\ dx - \int_0^1 \Psi uv'\ dx = \int_0^1 fv\ dx$$

b) Follow the procedure outlined in Sections 1.4-1.5 of the course notes and arrive at the system of linear algebraic equations $\mathbf{A}\hat{\mathbf{u}} = \mathbf{b}$, corresponding to the finite element discretization of variational problem (1.35). Write down the expressions for the elemental stiffness matrices and right hand sides. Is the resulting matrix $\mathbf{A}$ symmetric?

We proceed from

$$\int_0^1 \epsilon\ v'u'\ dx - \int_0^1 \Psi u v'\ dx = \int_0^1 fv\ dx \quad (Q)$$

The boundary points, tells us $\hat{u}_1, \hat{u}_3$ so we only need to find $\hat{u}_2$. Now letting $v = N$ and substituting $\hat{u}(x)$ for $u(x)$ in Q we have

$$\int_0^1 \epsilon\ v'u'\ dx - \int_0^1 \Psi u v'\ dx = \int_0^1 \epsilon\hat{u}'N_2'\ dx - \int_0^1 \Psi\hat{u}N_2'\ dx$$

Now using equation 1.20 and substituting in the $\hat{u}$ expression and only considering the first integral term we have

$$\int_0^1 \epsilon\hat{u}'N_2'\ dx = \epsilon\int_0^1 (\hat{u}_1N_1 + \hat{u}_2N_2 + \hat{u}_3N_3)'N_2'\ dx$$
$$= \epsilon\int_0^1 (\hat{u}_1N_1)'N_2' + (\hat{u}_2N_2)'N_2' + (\hat{u}_3N_3)'N_2'\ dx$$

And the second integral term

$$\Psi\int_0^1 (\hat{u}_1N_1 + \hat{u}_2N_2 + \hat{u}_3N_3)N_2'\ dx = \Psi\int_0^1 \hat{u}_1N_1N_2' + \hat{u}_2N_2N_2' + \hat{u}_3N_3N_2'\ dx$$

Now adding the two terms together and factoring for $\hat{u}$

$$\begin{aligned}
&\epsilon\int_0^1 (\hat{u}_1N_1)'N_2' + (\hat{u}_2N_2)'N_2' + (\hat{u}_3N_3)'N_2'\ dx - \Psi\int_0^1 \hat{u}_1N_1N_2' + \hat{u}_2N_2N_2' + \hat{u}_3N_3N_2'\ dx \\
&= \int_0^1 \epsilon(\hat{u}_1N_1)'N_2' - \Psi\hat{u}_1N_1N_2'\ dx + \int_0^1 \epsilon(\hat{u}_2N_2)'N_2' - \Psi\hat{u}_2N_2N_2'\ dx + \int_0^1 \epsilon(\hat{u}_3N_3)'N_2' - \Psi\hat{u}_3N_3N_2'\ dx \\
&= \hat{u}_1\int_0^1 \epsilon N_1'N_2' - \Psi N_1N_2'\ dx + \hat{u}_2\int_0^1 \epsilon N_2'N_2' - \Psi N_2N_2'\ dx + \hat{u}_3\int_0^1 \epsilon N_3'N_2' - \Psi N_3N_2'\ dx \\
&= a_{2,1}\hat{u}_1 + a_{2,2}\hat{u}_2 + a_{2,3}\hat{u}_3
\end{aligned}$$

Moving on the to the general case, the same arguments holds as used in the paragraph 1.24, so we have $\sum_{j=1}^{M} a_{i,j}\hat{u}_j, \quad i = 2, 3, \dots, M-1$.

$$a_{i,j} = \int_0^1 (\epsilon N_i'N_j' - \Psi N_i'N_j)\ dx$$

Where i for N is the basis test function $v(x)$ and j is the basis trial function in $\sum_j \hat{u}_jN_j$
Using 1.25 the global basis function for coefficients between $2, \dots M-1$ can be written as

$$
\begin{aligned}
a_{i,i-1} &= \int_{x_{i-1}}^{x_i} \epsilon N_i' N_{i-1}' - \Psi N_i' N_{i-1}\ dx \\
a_{i,i} &= \int_{x_{i-1}}^{x_i} \epsilon (N_i')^2 - \Psi N_i' N_i\ dx + \int_{x_i}^{x_{i+1}} \epsilon (N_i')^2 - \Psi N_i' N_i\ dx \\
a_{i,i+1} &= \int_{x_i}^{x_{i+1}} \epsilon N_i' N_{i+1}' - \Psi N_i' N_{i+1}\ dx
\end{aligned}
$$

going one step further, we can write them as local finite elements, where, like the book, we write the superscript from the perspective of the a coefficient in question

$$
\begin{aligned}
a_{i,i-1} &= \int_{x_{i-1}}^{x_i} \epsilon (N_2^{(i-1)})'(N_1^{(i-1)})' - \Psi (N_2^{(i-1)})' N_1^{(i-1)}\ dx \\
a_{i,i} &= \int_{x_{i-1}}^{x_i} \epsilon (N_2^{(i-1)})'\ ^2 - \Psi (N_2^{(i-1)})' N_2^{(i-1)} + \int_{x_i}^{x_{i+1}} \epsilon (N_1^{(i)})'\ ^2 - (\Psi N_1^{(i)})' N_1^{(i)} \\
a_{i,i+1} &= \int_{x_i}^{x_{i+1}} \epsilon (N_1^{(i)})'(N_2^{(i)})' - \Psi (N_1^{(i)})' N_2^{(i)}\ dx
\end{aligned}
$$

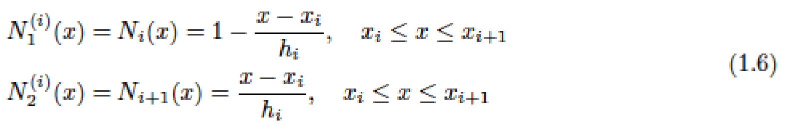
We have

$$
\kappa^{(i)} = \begin{bmatrix} k_{1,1}^{(i)} & k_{1,2}^{(i)} \\ k_{2,1}^{(i)} & k_{2,2}^{(i)} \end{bmatrix}
$$

using the general $a_{i,j}$ we can generalize the k expressions as

$$
k_{r,s}^{(i)} = \int_{x_i}^{x_{i+1}} [\epsilon (N_r^{(i)})'(N_s^{(i)})' - \Psi (N_r^{(i)})' N_s^{(i)}]\ dx
$$

To evaluate these limits we follow the same procedure as we did in 1.1.a, first note that eq.1.6 states



$$
\begin{aligned}
N_1^{(i)}(x) &= N_i(x) = 1 - \frac{x - x_i}{h_i}, \quad x_i \le x \le x_{i+1} \\
N_2^{(i)}(x) &= N_{i+1}(x) = \frac{x - x_i}{h_i}, \quad x_i \le x \le x_{i+1}
\end{aligned}
\tag{1.6}
$$

Now let $y_i(x) = \frac{x - x_i}{h_i}$, then $dy = \ \frac{1}{h_i} dx \Rightarrow h_i\ dy = \ dx$. The first limit is then $y_i(x_i) = \frac{x_i - x_i}{h_i} = 0$ and the second limit is $y_i(x_{i+1}) = \frac{x_{i+1} - x_i}{h_i} = \frac{h_i}{h_i} = 1$.

Additionally we have $\frac{d}{dx} N_1^{(i)} = \frac{1}{h} \frac{d}{dy} N_1^{(i)}$. We can now evaluate each of the terms in $\kappa^{(i)}$.

$$
\begin{aligned}
k^{(i)}_{1,1} &= \int_{x_i}^{x_{i+1}} [\epsilon(N_1^{(i)})'(N_1^{(i)})' - \Psi(N_1^{(i)})'N_1^{(i)}] \ dx \\
&= h\int_0^1 [\epsilon(\frac{1}{h}\frac{d}{dy}N_1^{(i)})(\frac{1}{h}\frac{d}{dy}N_1^{(i)}) - \Psi(\frac{1}{h}\frac{d}{dy}N_1^{(i)})N_1^{(i)}] \ dy \\
&= \frac{\epsilon}{h}\int_0^1 \frac{d}{dy}(1-y)\frac{d}{dy}(1-y) \ dy - \Psi\int_0^1 \frac{d}{dy}[1-y](1-y) \ dy \\
&= \frac{\epsilon}{h}\int_0^1 \ dx + \Psi\int_0^1 (1-y) \ dy \\
&= \frac{\epsilon}{h}[x]_0^1 + \Psi(\int_0^1 \ dy - \int_0^1 y \ dy) \\
&= \frac{\epsilon}{h} + \Psi([y]_0^1 - \frac{1}{2}[y^2]_0^1) \\
&= \frac{\epsilon}{h} + \frac{1}{2}\Psi
\end{aligned}
$$

$$
\begin{aligned}
k^{(i)}_{2,1} &= \int_{x_i}^{x_{i+1}} [\epsilon(N_2^{(i)})'(N_1^{(i)})' - \Psi(N_2^{(i)})'N_1^{(i)}] \ dx \\
&= h\int_0^1 \epsilon(\frac{1}{h}\frac{d}{dy}N_2^{(i)})(\frac{1}{h}\frac{d}{dy}N_1^{(i)}) - \Psi(\frac{1}{h}\frac{d}{dy}N_2^{(i)})N_1^{(i)} \ dy \\
&= \frac{\epsilon}{h}\int_0^1 \frac{d}{dy}(y)\frac{d}{dy}(1-y) \ dy - \Psi\int_0^1 (1-y)\frac{d}{dy}(y) \ dy \\
&= -\frac{\epsilon}{h}\int_0^1 \ dy - \Psi(\int_0^1 \ dy - \int_0^1 y \ dy) \\
&= -\frac{\epsilon}{h} - \Psi + \frac{1}{2}\Psi \\
&= -\frac{\epsilon}{h} - \frac{1}{2}\Psi
\end{aligned}
$$

$$
\begin{aligned}
k_{1,2} &= \int_{x_i}^{x_{i+1}} [\epsilon(N_1^{(i)})'(N_2^{(i)})' - \Psi(N_1^{(i)})'N_2^{(i)}] \ dx \\
&= h\int_0^1 \epsilon(\frac{1}{h}\frac{d}{dy}N_1^{(i)})(\frac{1}{h}\frac{d}{dy}N_2^{(i)}) - \Psi(\frac{1}{h}\frac{d}{dy}N_1^{(i)})N_2^{(i)} \ dy \\
&= \frac{\epsilon}{h}\int_0^1 \frac{d}{dy}(1-y)\frac{d}{dy}(y) \ dy - \Psi\int_0^1 \frac{d}{dy}[(1-y)]y \ dy \\
&= \frac{\epsilon}{h}\int_0^1 \frac{d}{dy}(y)\frac{d}{dy}(1-y) \ dy - \Psi\int_0^1 (-1)y \ dy \\
&= -\frac{\epsilon}{h} + \frac{1}{2}\Psi
\end{aligned}
$$

And lastly

$$\begin{aligned}
k_{2,2} &= \int_{x_i}^{x_{i+1}} [\epsilon(N_2^{(i)})'(N_2^{(i)})' - \Psi(N_2^{(i)})'N_2^{(i)}] \ dx \\
&= h\int_0^1 \epsilon(\frac{1}{h}\frac{d}{dy}N_2^{(i)})(\frac{1}{h}\frac{d}{dy}N_2^{(i)}) - \Psi(\frac{1}{h}\frac{d}{dy}N_2^{(i)})N_2^{(i)} \ dx \\
&= \frac{\epsilon}{h}\int_0^1 \ dy - \Psi\int_0^1 y \ dy \\
&= \frac{\epsilon}{h} - \frac{1}{2}\Psi
\end{aligned}$$

Combining the results we have

$$\kappa^{(i)} = \begin{bmatrix} \frac{\epsilon}{h_i} + \frac{1}{2}\Psi & -\frac{\epsilon}{h_i} + \frac{1}{2}\Psi \\ -\frac{\epsilon}{h_i} - \frac{1}{2}\Psi & \frac{\epsilon}{h_i} - \frac{1}{2}\Psi \end{bmatrix}$$

Since $\kappa^{(i)}$ is the elements of the stiffness matrices this answers the first question. For the second question about symmetry, it follows from $\kappa^{(i)}$ that A cannot be a symmetric matrix as $k_{1,2} \neq k_{2,1}$ and hence we have two different elements above and below the matrix, respectively.

This yields all the relevant components, for the sake of brevity I will not write the A matrix and u vector out. Instead just see them as described above eq1.30, simply insert the appropriate $\kappa^{(i)}$ values in A.

c) If the matrix $\mathbf{A}$ is *positive definite*, that is, $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} > 0$ for all non-zero vectors $\mathbf{v}$, then it is necessarily non-singular and therefore the linear algebraic system $\mathbf{A}\hat{\mathbf{u}} = \mathbf{b}$ admits a unique solution for every right-hand side $\mathbf{b}$.

   Show that $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} > 0$ for every vector $\mathbf{v}$, such that $\mathbf{v}_1 = \mathbf{v}_M = 0$ and thus the linear algebraic system resulting from (1.35) always admits a solution.

   Note, that you have to show *two* things: (i) $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} \geq 0$ for every vector $\mathbf{v}$ satisfying the boundary conditions; and (ii) $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} = 0$ implies $\mathbf{v} = \mathbf{0}$.

   The following hints are given: $\mathbf{v}^{\mathrm{T}}\mathbf{A}\mathbf{v} = a(\sum_{i=1}^{M} v_i N_i, \sum_{j=1}^{M} v_j N_j)$; $\int v v' \mathrm{d}x = 1/2 \int (v^2)' \mathrm{d}x$.

I did not solve this.

d) Let $f = 1$ (and $\Psi \neq 0$); the analytical solution to (1.34) in this case is

$$u(x) = \frac{1}{\Psi}\left(\frac{1 + (\exp(\Psi/\epsilon) - 1)x - \exp(x\Psi/\epsilon)}{\exp(\Psi/\epsilon) - 1}\right)$$

Plot the solution for fixed $\Psi = 1$ and various values of $\epsilon \in \{1, 0.01, 0.0001\}$.

We plot it in Maple

```
psi := 1;
                              psi := 1

eps1 := 1;
eps2 := 0.01;
eps3 := 0.0001;
u := (x, eps) -> (1 + (exp(psi/eps) - 1)*x - exp(x*psi/eps))/(psi*(exp(psi/eps) - 1));
u := proc (x, eps) options operator, arrow; (1+(exp(psi/eps)-1)*\

  x-exp(x*psi/eps))/(psi*(exp(psi/eps)-1)) end proc


plot([u(x, eps1), u(x, eps2), u(x, eps3)], x = 0 .. 1, color = ["Red", "Blue", "Green"], lege
```

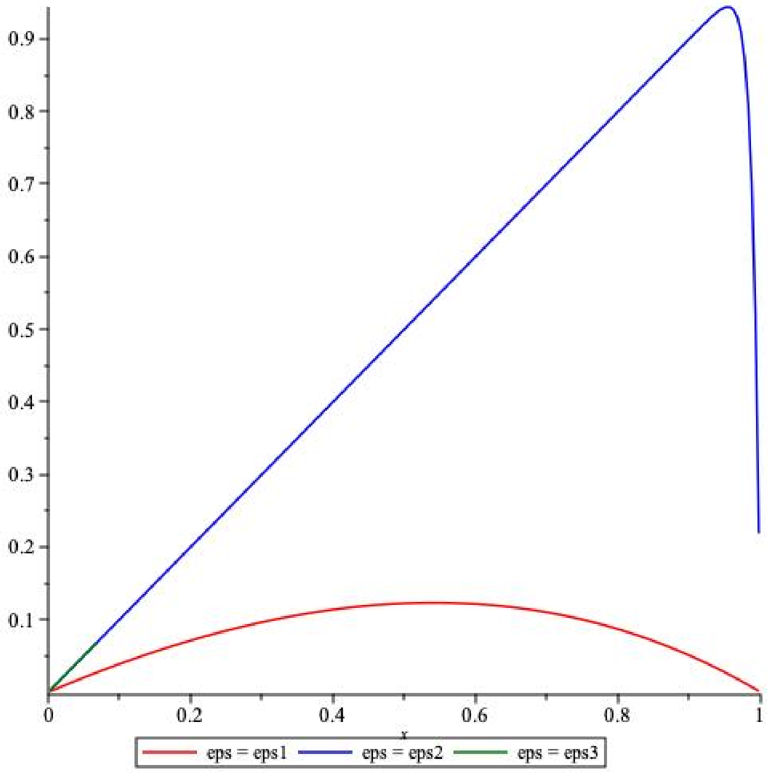


Figure 1: Resulting plot from the maple code

e) Make appropriate modifications to the program you have written for solving Exercise 1.2, so that is solves the boundary value problem (1.35) instead and verify its convergence. Use the same parameters as in step d). Comment on the variation of the quality of the computed FEM solution with ε.

From algorithm2 we node that $b[2] = b[2] - k_{2,1}^{(1)} * c$ and from exercise1.5.a we know that the RHS of 1.35 and assuming $x_{i-1} - x_i = x_{i+1} - x_i = h$ we can use $y_i(x) = \frac{x-x_i}{h_i}$ as subtitution to find an expression for RHS.

$$\begin{aligned}
\int_{x_{i-1}}^{x_{i+1}} N_i(x)\,dx &= \int_{x_{i-1}}^{x_i} N_i(x)\,dx + \int_{x_i}^{x_{i+1}} N_{i+1}(x)\,dx \quad (f=1) \\
&= \int_{x_{i-1}}^{x_i} \frac{x-x_{i-1}}{h_i}\,dx + \int_{x_i}^{x_{i+1}} 1 - \frac{x-x_i}{h_i}\,dx
\end{aligned}$$

Let $y_i(x) = \frac{x-x_i}{h_i}$ and use equidsitance s.t. $h_i = h_{i-1}$ then

$$\begin{aligned}
&\int_{x_{i-1}}^{x_i} \frac{x-x_{i-1}}{h_i}\,dx + \int_{x_i}^{x_{i+1}} 1 - \frac{x-x_i}{h_i}\,dx \\
&= h\int_0^1 y_{i-1}\,dy + h\int_0^1 y_i\,dy \\
&= \frac{h}{2}[y_{i-1}^2]_0^1 + \frac{h}{2}[y_i^2]_0^1 \\
&= \frac{h}{2}[(\frac{x-x_{i-1}}{h_i})^2]_0^1 + \frac{h}{2}[(\frac{x-x_i}{h_i})^2]_0^1 \\
&= \frac{h}{2}[(\frac{h}{h})^2]_0^1 + \frac{h}{2}[(\frac{h}{h})^2]_0^1 \\
&= \frac{h}{2} + \frac{h}{2} = h \\
&\Rightarrow \int_0^1 fv\,dx = fh
\end{aligned}$$

The code is then

```
#Exercise 1.5.e
function assembly(M,h,eps, Psi,c)
    A = zeros(M,M)
    b = zeros(M,1)


    for i = 1:1:M-1
        k1_1 = eps/h + 1/2* Psi
        k1_2 = -eps/h + 1/2*Psi
        k2_1 = -eps/h - 1/2*Psi
        k2_2 = eps/h - 1/2*Psi

        A[i,i] = A[i,i] + k1_1
        A[i,i+1] = k1_2
        A[i+1,i+1] = k2_2
        A[i+1, i] = k2_1

        b[i] = h - k2_1*c
    end
    return A,b
end


function boundary_conditions(A,b,M,c,d)
    b[1] = c
    b[2] = b[2]-A[2,1]*c

    A[1,1] = 1.0
    A[1,2] = 0.0

    b[M] = d
    b[M-1] = b[M-1] - A[M-1,M]*d

    A[M,M] = 1.0
    A[M-1,M] = 0.0
    A[M, M-1] = 0.0
    return A,b
end

function conv_rate1_5_e(iters, L,eps, Psi)
    M_values = Float64[]
    errors = Float64[]
    h_sq_values = Float64[]
    c = 0 #start boundary
    d = 0 #end boundary

    for M in range(3,iters)
        x = range(0,L, length=M)
        h = x[2]-x[1] #assumes equidistance between points

        A,b = assembly(M,h,eps, Psi,c)
        A,b = boundary_conditions(A,b,M,c,d)

        uhat = A \ b

        error_term = maximum(abs.(u_anal(x,eps, Psi) - uhat))
        push!(M_values, M)
        push!(errors, error_term)
        push!(h_sq_values, h[1] * h[1])

        #Plot best mesh
        if M == last(iters)
            q = plot(x, uhat,
                linestyle=:dash,
                label="approx solution",
                color=:blue)
            savefig(q, "w1/1_5_e_approxsol_plot_eps_$(eps).png")
        end
    end

    p = plot(M_values, errors,
             xaxis=:log, yaxis=:log,
             marker=:circle,
             title="convergence rate",
             xlabel="M",
             ylabel="Max Error",
             label="FEM Error")

    plot!(p, M_values, 20000 .* h_sq_values,
        linestyle=:dash,
        label="2000 * h^2 Reference",
        color=:red)

    # q = plot(M_values, uhat_vals,
    #     linestyle=:dash,
    #     label="approx solution",
    #     color=:blue)


    savefig(p, "w1/1_5_e_convergence_plot_eps_$(eps).png")

    # display(p)


    #Get the slop between the first two points
    s_start = (log(errors[2]) - log(errors[1])) / (log(M_values[2]) - log(M_values[1]))
    #Get the slop between last two points (we expect approx -2.0)
    s_end   = (log(errors[end]) - log(errors[end-1])) / (log(M_values[end]) - log(M_values[en

    println("Convergence Rate (First 2 points): ", s_start)
    println("Convergence Rate (Last 2 points):  ", s_end)
end


Psi = 1.0
conv_rate1_5_e(1000, 1, 1.0, Psi)
conv_rate1_5_e(1000, 1, 0.01, Psi)
conv_rate1_5_e(1000, 1, 0.0001, Psi)
```

Which produces the plots

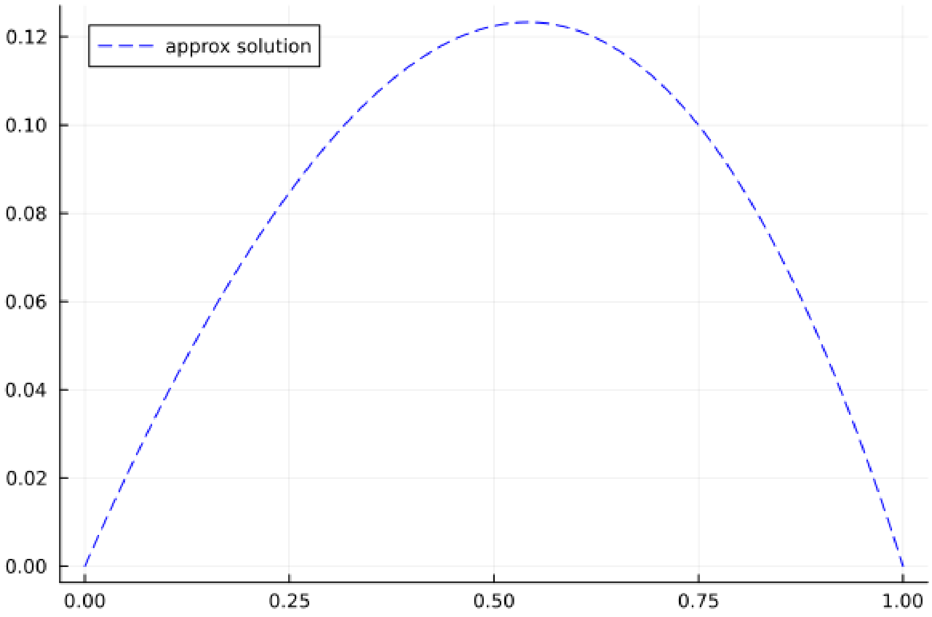


Figure 2: $\epsilon = 1.0$

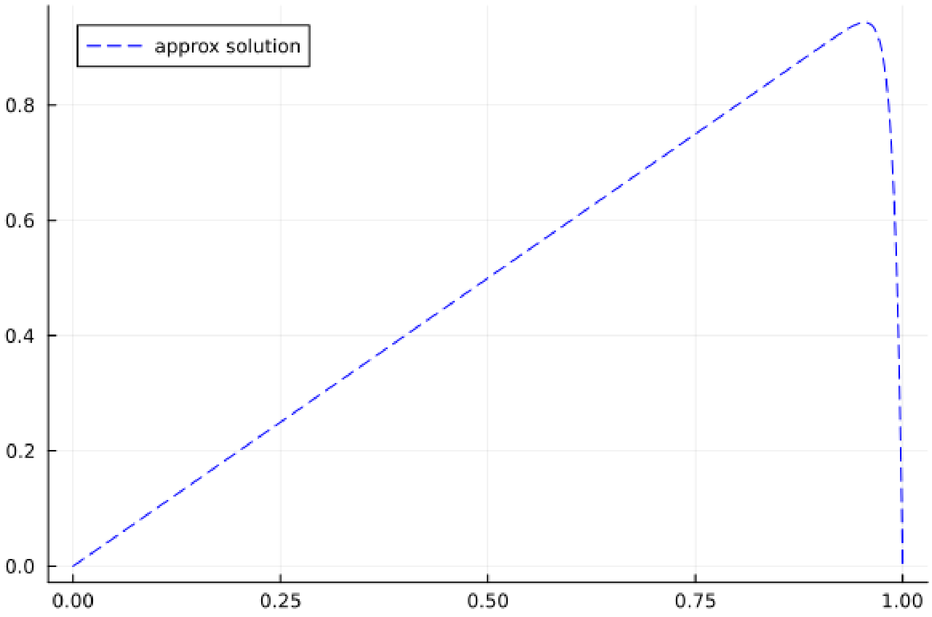


Figure 3: $\epsilon = 0.01$

4 Exercise 1.6

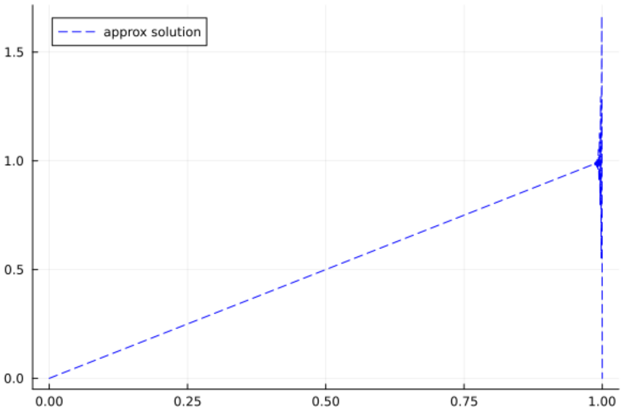


Figure 4: $\epsilon = 0.0001$

Exercise 1.6

In this exercise the goal is to develop an algorithm that can be used for automatic Adaptive Mesh Refinement (AMR). Such an algorithm consists of two ingredients, namely, a method for refining the mesh and a method for selecting which mesh elements to be refined. The method for refining the mesh will be based on h-type refinement where existing elements are subdivided into two new element of equal size to reduce errors locally. The selection of elements for refinement will be based on assuming that an optimal mesh have errors distributed equally across elements of the mesh.

In the following, we will consider how to automatically adapt a mesh to the function

$$u(x) = e^{-800(x-0.4)^2} + 0.25e^{-40(x-0.8)^2}$$

across the interval $0 \leq x \leq 1$ such that an interpolating piece-wise polynomial (1.7) achieves a user-defined accuracy level for the accurate representation of $u(x)$.

a) A simple way to guide the selection of mesh elements for refinement, is to estimate the error decrease rate that will result if the i'th element is subdivided in two smaller elements of equal size. We will employ a metric which measures the change in approximate solution across elements by refinement. The metric is defined in terms of the L_2 (mean-square) norm as

$$\Delta err_i = ||\Pi_{h/2,i}u - \Pi_{h,i}u||_{L_2(e_i)}, \quad ||f||_{L_2(\Omega)} \equiv \left(\int_\Omega |f|^2 dx\right)^{1/2}$$

$\Pi_{h,i}u$ is an operator which defines the piece-wise interpolation of $u(x)$ onto the i'th element with mesh size h (see equation (1.7)). How can the metric be computed? Detail this. Then, write a Matlab routine which can compute the estimated error decrease rate based on this metric. The estimate should be based on the mesh tables `VX` and `EToV` (learn about such tables in beginning of Chapter 2) which defines a coarse mesh. `err` is a vector with K elements such that `err(i)` is the estimated error for the i'th element in the coarse mesh.

Head:
```
function [err] = compute_error_decrease(fun,VX,EToV);
```

Consider an arbitrary line segment with three points $e_i = [x_i, x_m, x_{i+1}]$. For the L^2-norm we know that

$$f = \Pi_{h/2,i}u - \Pi_{h,i}u$$

Define the piece-wise function

$$\Pi_{h/2,i}u = \begin{cases} a_1x + b_1 & x_i \leq x \leq x_m \\ a_2x + b_2 & x_m \leq x \leq x_{i+1} \end{cases}$$

and the function

$$\Pi_{h,i}u = a_3x + b_3 \quad x_i \leq x \leq x_{i+1}$$

Ignoring the square root, the integral for the L^2 function is

$$\begin{aligned}
&\int_{x_i}^{x_{i+1}} |\Pi_{h/2,i}u - \Pi_{h,i}u|^2 = \int_{x_i}^{x_m} |\Pi_{h/2,i}u - \Pi_{h,i}u|^2 \; dx + \int_{x_m}^{x_{i+1}} |\Pi_{h/2,i}u - \Pi_{h,i}u|^2 \; dx \\
&= \int_{x_i}^{x_m} |a_1x + b_1 - a_3x - b_3|^2 \; dx + \int_{x_m}^{x_{i+1}} |a_2x + b_2 - a_3x - b_3|^2 \; dx \\
&= \int_{x_i}^{x_m} |(a_1 - a_3)x + b_1 - b_3|^2 \; dx + \int_{x_m}^{x_{i+1}} |(a_2 - a_3)x + b_2 - b_3|^2 \; dx \\
&\Rightarrow ||f||_{L_2([x_i, x_m, x_{i+1}])} = (\int_{x_i}^{x_m} |(a_1 - a_3)x + b_1 - b_3|^2 \; dx + \int_{x_m}^{x_{i+1}} |(a_2 - a_3)x + b_2 - b_3|^2 \; dx)^{1/2}
\end{aligned}$$

Where coefficients are

$$a_1 = \frac{y_m - y_i}{x_m - x_i}$$
$$b_1 = y_i - a_1x_i$$

And

$$a_2 = \frac{y_{i+1} - y_m}{x_{i+1} - x_m}$$
$$b_2 = y_m - a_2x_m$$

And

$$a_3 = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$
$$b_3 = y_i - a_3x_i$$

Notice that these are second degree polynomials and we can solve these analytically. let $A_{1,3} = a_1 - a_3$, $B_{1,3} = b_1 - b_3$, then $A_{1,3}x + B_{1,3} \in \mathbb{R} \Rightarrow |A_{1,3}x + B_{1,3}|^2 = (A_{1,3}x + B_{1,3})^2$ and so we can evaluate the two integrals. Starting with the first integral we have

$$\int_{x_i}^{x_m} |(a_1 - a_3)x + b_1 - b_3|^2 \ dx = \int_{x_i}^{x_m} |A_{1,3}x + B_{1,3}|^2 \ dx$$
$$= \int_{x_i}^{x_m} (A_{1,3}x + B_{1,3})^2 \ dx = \int_{x_i}^{x_m} (A_{1,3}x)^2 + 2A_{1,3}B_{1,3}x + B^2 \ dx$$
$$= A_{1,3}^2 \int_{x_i}^{x_m} x^2 \ dx + 2A_{1,3}B_{1,3} \int_{x_i}^{x_m} x \ dx + B_{1,3}^2 \int_{x_i}^{x_m} \ dx$$
$$= \frac{A_{1,3}^2}{3}[x^3]_{x_i}^{x_m} + \frac{2A_{1,3}B_{1,3}}{2}[x^2]_{x_i}^{x_m} + B_{1,3}^2[x]_{x_i}^{x_m}$$
$$= \frac{A_{1,3}^2}{3}[x_m^3 - x_i^3] + A_{1,3}B_{1,3}[x_m^2 - x_i^2] + B_{1,3}^2[x_m - x_i]$$

For the second integral in the L^2-norm we use the same strategy with $A_{2,3} = a_2 - a_3$ and $B_{2,3} = b_2 - b_3$,

$$\int_{x_m}^{x_{i+1}} |(a_2 - a_3)x + b_2 - b_3|^2 \ dx)^{1/2} = \int_{x_m}^{x_{i+1}} (A_{2,3}x + B_{2,3})^2 \ dx$$
$$= \int_{x_m}^{x_{i+1}} A_{2,3}^2x^2 + 2A_{2,3}B_{2,3}x + B_{2,3}^2 \ dx$$
$$= A_{2,3}^2 \int_{x_m}^{x_{i+1}} x^2 \ dx + 2A_{2,3}B_{2,3} \int_{x_m}^{x_{i+1}} x \ dx + B_{2,3}^2 \int_{x_m}^{x_{i+1}} \ dx$$
$$= \frac{A_{2,3}^2}{3}[x^3]_{x_m}^{x_{i+1}} + A_{2,3}B_{2,3}[x^2]_{x_m}^{x_{i+1}} + B_{2,3}^2[x]_{x_m}^{x_{i+1}}$$
$$= \frac{A_{2,3}^2}{3}[(x_{i+1})^3 - x_m^3] + A_{2,3}B_{2,3}[x_{i+1}^2 - x_m^2] + B_{2,3}^2[x_{i+1} - x_m]$$

Putting the two terms together we have L^2-norm, note there was not enough space on 1 line so I had to put it on 2 lines

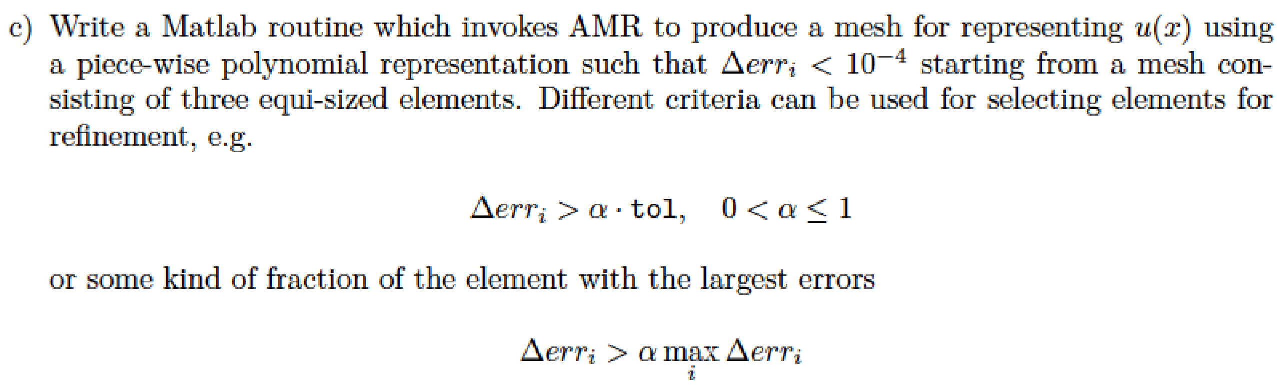
$$||f||_{L_2([x_i,x_m,x_{i+1}])} = [\int_{x_i}^{x_m} |(a_1 - a_3)x + b_1 - b_3|^2 \ dx + \int_{x_m}^{x_{i+1}} |(a_2 - a_3)x + b_2 - b_3|^2 \ dx]^{1/2}$$
$$= [(\frac{A_{1,3}^2}{3}[x_m^3 - x_i^3] + A_{1,3}B_{1,3}[x_m^2 - x_i^2] + B_{1,3}^2[x_m - x_i])$$
$$+ \frac{A_{2,3}^2}{3}[(x_{i+1})^3 - x_m^3] + A_{2,3}B_{2,3}[x_{i+1}^2 - x_m^2] + B_{2,3}^2[x_{i+1} - x_m]]^{1/2}$$

b) Write a Matlab routine which can subdivide a coarse mesh and produce a refined mesh by refining only elements which has been marked for refinement using an element index vector `idxMarked`.

   Head:
   `function [EToVfine,xfine] = refine_marked(EToVcoarse,xcoarse,idxMarked)`

Explanation here

```
#Exercise 1.6.b
function refine_marked(EToVcoarse, xcoarse, idxMarked)
    #OKAY I FIGURED OUT WAHT THESE TABLES ARE
    #xcoarse = VX which is the points, e.g. x_1,x_2,..x_i,...x_n
    #EToVcoarse is the ELEMENTS, that is the linepieces between the points!!!
    #idxMarked is an element that is marked for refinement!

    #Find number of marks
    idxMarked_set = Set(idxMarked) #lookup is O(1) so faster later
    new_addtions = length(idxMarked_set) #every new refined point "xm" will yield a new linep

    #--- Setup VX Table ---
    n_vx_old = length(xcoarse) #No of points.


    #Number of new VX elements
    # vx_new_elems = vx_old_elems + no_of_new_elems
    vx_points_new = n_vx_old + new_addtions

    #Instantiation
    vx_new = zeros(Float64, vx_points_new)
    vx_new[1:n_vx_old] = xcoarse #set all old points


    #--- Setup EToV ---
    n_elem_old = size(EToVcoarse,1) #Number of elements
    EToV_elements_new = n_elem_old + new_addtions
    EToV_new = zeros(Int, EToV_elements_new,2) #we need the connect from the left side and ri

    #-- Setup Etc. ---
    vx_cnt = n_vx_old + 1 #fill in points at the end of the old points
    elem_cnt = 1  #fill elements from beginning

    #We are going to iterate through all the elements and look to see which needs refinement!
    for elem_id in 1:1:n_elem_old

        #find points connected to element
        i_left = EToVcoarse[elem_id, 1] #"ei" left connection to point
        i_right = EToVcoarse[elem_id, 2] #"ei" right connection to point

        if elem_id in idxMarked_set

            #get points connected to element
            xi = xcoarse[i_left]  #left point
            xip1 = xcoarse[i_right] #right point

            #Add new elemnt to VX
            xm = (xi+xip1) / 2.0 #new x point
            vx_new[vx_cnt] = xm #add to new point "xm" VX table

            i_mid = vx_cnt #new point in VX

            #Add new "em" element connections to EToV
            EToV_new[elem_cnt, 1] = i_left #set left side of "em" to the previous left connec
            EToV_new[elem_cnt, 2] = i_mid #set the right side of "em" to the new point "xm"
            elem_cnt += 1 #next row in EToV

            #Readjust old "ei" to be next element in EToV
            #Note that we need to reset all point connections to old "ei" because it has been
            EToV_new[elem_cnt, 1] = i_mid #set "ei" left connection to "xm"
            EToV_new[elem_cnt, 2] = i_right #set "ei" to the right connection
            elem_cnt += 1 #next row in EToV
            vx_cnt += 1 #next row in VX table

        else
            #--- Keep The element unrefned ---
            EToV_new[elem_cnt,1] = i_left #element remains connected to same left point
            EToV_new[elem_cnt,2] = i_right #element remains conncected to same right point
            #Notice that we do not move the VX row because this will be the next refinement p
            elem_cnt += 1
        end

    end

    return vx_new, EToV_new
end
```



c) Write a Matlab routine which invokes AMR to produce a mesh for representing $u(x)$ using a piece-wise polynomial representation such that $\Delta err_i < 10^{-4}$ starting from a mesh consisting of three equi-sized elements. Different criteria can be used for selecting elements for refinement, e.g.

$$\Delta err_i > \alpha \cdot \texttt{tol}, \quad 0 < \alpha \leq 1$$

or some kind of fraction of the element with the largest errors

$$\Delta err_i > \alpha \max_i \Delta err_i$$

We do as requested in julia

```julia
#exercise 1.6.c
function AMR(u, tol, VX, EToV, max_iter)
    history_dofs = []
    history_err = []

    for iter in 1:1:max_iter
        #use 1.6.a to compute error
        err_vec = compute_error_decrease(u, VX, EToV)
        max_err = maximum(err_vec) #NB! this is NOT the 1 norm. We can do this because we fou

        push!(history_dofs, length(VX))
        push!(history_err, max_err)

        if max_err < tol #the maximum is below the threshold
            break
        end

        idxMarks = findall(elem -> elem > tol, err_vec) #I go HARD and I take all the elmeent
        VX, EToV = refine_marked(EToV, VX, idxMarks)
    end

    return VX, EToV, history_dofs, history_err #return the refined points and elements table
end
```

d) Present plots of the computed solution, the element size distribution for the final mesh and for evaluating the performance of the algorithm that are used in c) (e.g. error vs. mesh points).

Here is the generated code

```
using Plots

function plot_amr_results(VX, EToV, dofs, errs, u_exact)
    x_plot = range(minimum(VX), maximum(VX), length=1000)
    y_exact = u_exact.(x_plot)

    # Sort the unstructured nodes so the line plot connects them correctly
    perm = sortperm(VX)
    sorted_VX = VX[perm]
    sorted_u = u_exact.(sorted_VX) # Evaluation at nodes

    p1 = plot(x_plot, y_exact,
        label="Exact u(x)",
        lw=2, color=:black,
        title="Computed vs Exact Solution",
        xlabel="x", ylabel="u(x)")

    # Plot the nodes as red dots
    plot!(p1, sorted_VX, sorted_u,
        label="AMR Solution",
        seriestype=:scatter, markersize=3, color=:red)

    # Connect nodes with dashed lines to show the piecewise linear approximation
    plot!(p1, sorted_VX, sorted_u,
        label="",
        lw=1, linestyle=:dash, color=:red)

    #Plot 2 Element Size Distribution
    # Calculate element sizes h = |x_R - x_L|
    n_elems = size(EToV, 1)
    element_sizes = zeros(n_elems)

    for i in 1:n_elems
        idx_L = EToV[i, 1]
        idx_R = EToV[i, 2]
        element_sizes[i] = abs(VX[idx_R] - VX[idx_L])
    end

    p2 = histogram(element_sizes,
        bins=20,
        title="Element Size Distribution",
        xlabel="Element Size (h)",
        ylabel="Frequency",
        legend=false, color=:blue)

    # --- Plot 3: Convergence (Error vs DOFs) ---
    p3 = plot(dofs, errs,
        xaxis=:log, yaxis=:log,
        marker=:circle,
        title="Convergence: Max Error vs DOFs",
        xlabel="Degrees of Freedom (Nodes)",
        ylabel="Max L2 Error",
        label="Error Decay",
        lw=2, color=:green)

    # --- Combine and Display ---
    final_plot = plot(p1, p2, p3, layout=(3, 1), size=(600, 900))
    display(final_plot)

    savefig(final_plot, "w1/1_6_d_final_plot.png")
end

u = x -> exp(-800*(x-0.4)^2) + 0.25*exp(-40*(x-0.8)^2)
VX = [0.0, 1/3, 2/3, 1.0]
EToV = [1 2; 2 3; 3 4]
tol = 1e-6
max_iter = 20

final_VX, final_EToV, history_dofs, history_err = AMR(u, tol, VX, EToV, max_iter)

plot_amr_results(final_VX, final_EToV, history_dofs, history_err, u)
```

Computed vs exact Looks visually pleasing, it seems to imitate the exact solution well.

Element size distribution It looks to be clustering around $(0.00, 0.5]$

(Convergence: Max Error vs DOFs we see taht the max error decreases as the degrees of freedom diminshes, this is sensible as increased degrees of freedom implies a finer mesh and a finer mesh is expected to give a better solution. Ideally this function should be a linear function with negative slope.

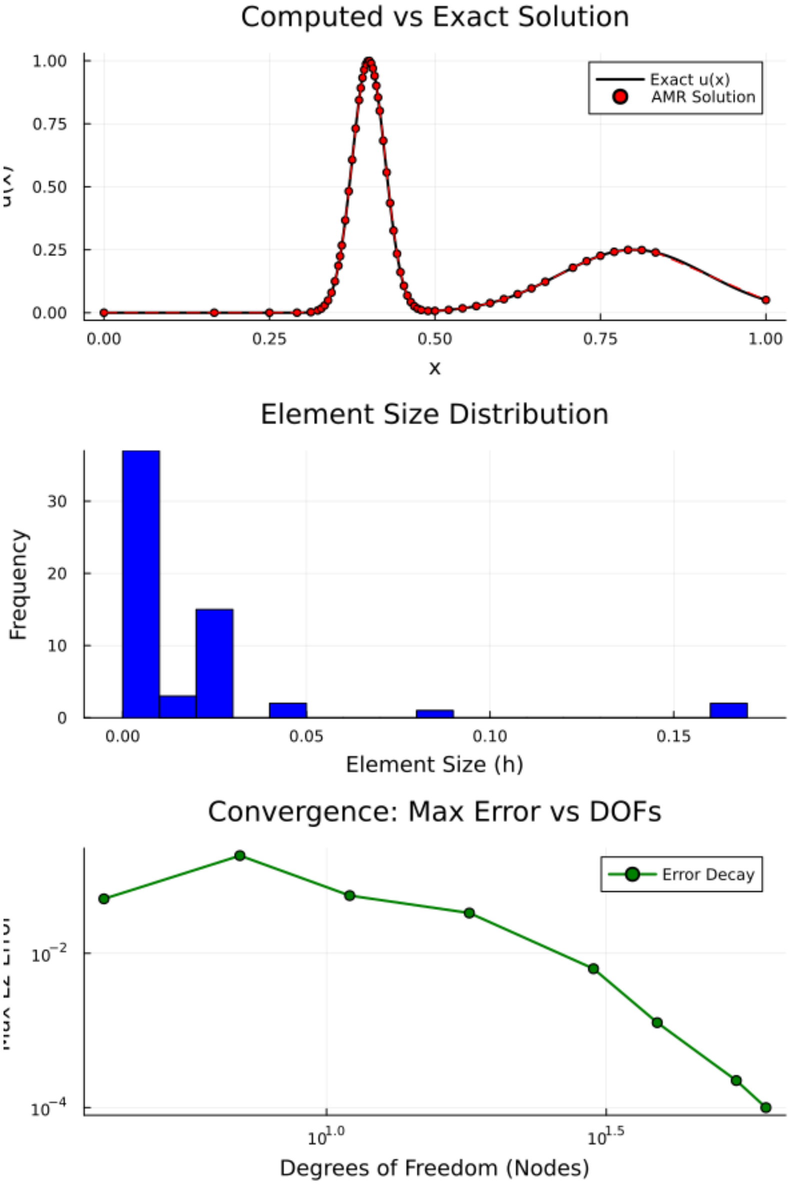


Figure 5: The computed plot from the code above

5 Exercise 1.7

By the assumption of u we have

$$\begin{aligned} u(x) =& \exp\left(-800(x-0.4)^2\right) + 0.25\exp\left(-40(x-0.8)^2\right) \\ u''(x) =& -1600\exp\left(-800(x-0.4)^2\right) \\ &+ (-1600x+640)^2\exp\left(-800(x-0.4)^2\right) \\ &- 20\exp\left(-40(x-0.8)^2\right) \end{aligned}$$

given $u''(x) - u(x) = f(x)$ we then have

Exercise 1.7

Consider the FEM applied to the following boundary value problem

$$u''(x) - u(x) = f(x), \quad 0 \le x \le 1, \quad u(0) = c, \quad u(1) = d$$

The goal of this exercise will be to write a Matlab program which can solve this boundary value problem using the classical FEM combined with an Adaptive Mesh Refinement (AMR) algorithm. The AMR should be based on h-refinement where the mesh elements are subdivided as in Exercise 1.6. This will require two basic ingredients, namely, a method for doing mesh refinements (reuse method from Exercise 1.6) and a way to estimate the errors (*a posteriori* where the computed FEM solution is used to estimate the accuracy) in order to be able to guide the refinements until a desired overall accuracy level has been achieved.

a) Determine $f(x)$, c and d using the Method of Manufactured solution where it is assumed that the exact solution is $u(x) = e^{-800(x-0.4)^2} + 0.25e^{-40(x-0.8)^2}$. Visualize this solution for $0 \le x \le 1$ and comment on expectations to an optimal mesh for the problem. It is not allowed to use the knowledge of this exact solution in the AMR procedure in this exercise.

$$\begin{aligned} f(x) &= \left(1600x^2 - 2560x + 1003.75\right) e^{-1.6(5x-4)^2} \\ &\quad + \left(2.56 \times 10^6 x^2 - 2.048 \times 10^6 x + 4.07999 \times 10^5\right) e^{-32(5x-2)^2} \end{aligned}$$

Plotting the solution we have

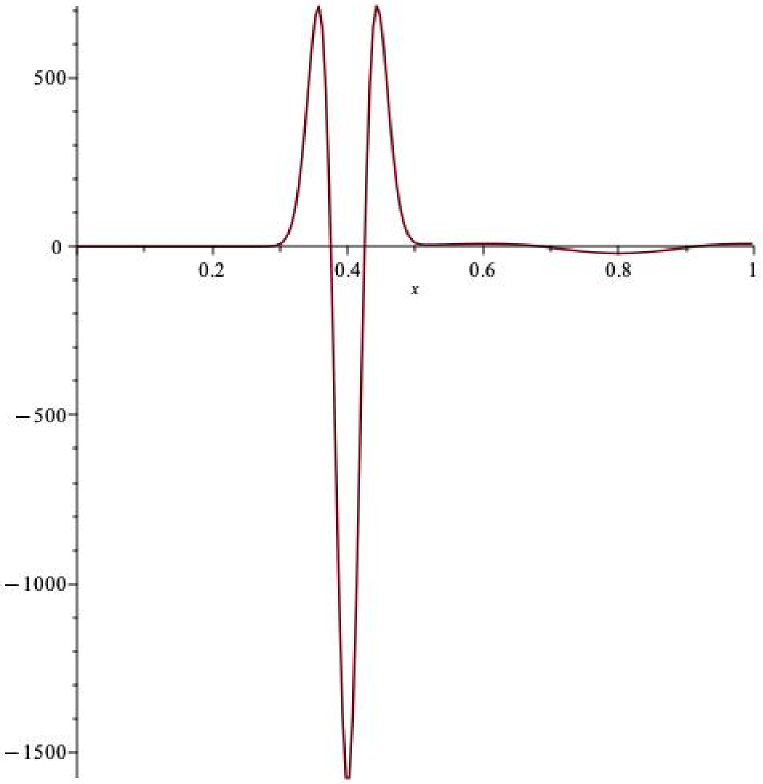


Figure 6: values of f with the assumed u

From the exact solution u figure 5 tells us we'll need refinement for $I_1 = (0.3, 0.5)$ and $I_2 =$

$(0.7, 0.9)$ where we expects I_1 to need much finer refinement than I_2.

b) Write a Matlab routine which can compute error estimates based on the L_2-norm on a per-element basis. The error estimate can be expressed in terms of computed solution values (`uhc` and `uhf`) obtained on a coarse (`xc`) and a refined mesh (`xf`). It is advantageous to take into account which elements that have been marked for refinement (`idxmarked`) and store information about how elements are then subdivided (`Old2New`) in appropriate arrays. If you see other ways of doing this setup feel free to do so.

Head: (suggestion)
`function [err] = errorestimate(xc,xf,uhc,uhf,EToVc,EToVf,Old2New)`

We first need to find an expression for the error estimate. From exercise1.6.a we know that f in the L^2-norm is the difference between the coarse mesh result and the fine mesh result, i.e. $f = \Pi_{h/2,i}u - \Pi_{h,i}u = u_{uhf} - u_{uhc}$. Additionally from 1.7.a we only want to integrate a subset $[0, h]$. The goal is to find the L^2-norm of $[0, h]$, that is, the part of the mesh we want to refine.

$$\Delta err_i^2 = ||\Pi_{h/2,i}u - \Pi_{h,i}u|| = \int_0^h |u_{uhf} - u_{uhc}|^2 \; dx)$$

since $u_{uhf} - u_{uhc} \in \mathbb{R}$ then $|u_{uhf} - u_{uhc}|^2 = (u_{uhf} - u_{uhc})^2$, write $d_\Delta = u_{uhf} - u_{uhc}$, i.e. the distance difference between the fine and course exact solution, then

$$\int_0^h |u_{uhf} - u_{uhc}|^2 \; dx = \int_0^h (u_{uhf} - u_{uhc})^2 \; dx = \int_0^h d_\Delta^2(x) \; dx \quad (D)$$

Define $r = \frac{x}{h}$ where $r \in [0, 1]$, where x is a point on the line and h is the length of the line piece. So r is a normalized line and $x = r \cdot h \Rightarrow dx = h \; dr$.

Let a_0 be the error value at the left node $x = 0$, that is $(u_{uhf}(0) - u_{uhc}(0))$ and let a_h be the error value at the right node $x = h$, that is $(u_{uhf}(h) - u_{uhc}(h))$. The linear interpolation line is then

$$d_\Delta(r) = a_0(1 - r) + a_h r$$

Using a change of variable we solve (D)

$$\begin{aligned}
&\int_0^h d_\Delta^2(x) \; dx = \int_0^1 (a_0(1 - r) + a_h r)^2 h \; dr \\
&= h \int_0^1 a_0^2(1 - r)^2 + 2a_0a_h(1 - r)(r) + a_h^2 r^2 \; dr \\
&= h[a_0^2 \int_0^1 (1 - r)^2 \; dr + 2a_0a_h \int_0^1 (1 - r)r \; dr + a_h^2 \int_0^r r^2 \; dr \\
&= h[\frac{a_0^2}{3} + \frac{a_0a_h}{3} + \frac{a_h^2}{3}] \quad \text{(maple used)} \\
&= \frac{h}{3}(a_0^2 + a_0a_h + a_h^2)
\end{aligned}$$

Hence the L^2-norm on $[0, h]$ is

$$\Delta err_i = (\int_0^h |u_{uhf} - u_{uhc}|^2 \; dx)^{1/2} = \sqrt{\frac{h}{3}(a_0^2 + a_0 a_h + a_h^2)}$$

Using this analytical solution we define the function to be

```
function L2_norm(a0,ah,h)
    return h/3 * (a0^2 + a0*ah + ah^2)
end

function errorestimate(xc, xf, uhc, uhf, EToVc, EToVf, Old2New)
    #Setup coarse
    c_elem_sz = size(EToVc, 1)

    #Setup fine
    f_elem_sz = size(EToVf, 1)

    #Setup miscellanous
    err = zeros(Float64, c_elem_sz) #we can only error estimates for as many coarse elements

    for e_id in 1:1:f_elem_sz

        #Coarse Info
        c_e_id = Old2New[e_id] #Get old element id

        #get coarse points
        c_idx_L = EToVc[c_e_id, 1] #left point of course elmeent
        c_idx_R = EToVc[c_e_id, 2] #right point of course element

        c_xL = xc[c_idx_L] #get left element
        c_xR = xc[c_idx_R] #get right element
        h_coarse = c_xR - c_xL #I am assuming that xL < xR becaues we are in 1D


        #get coarse function values
        c_uL = uhc[c_idx_L] #left function coarse value
        c_uR = uhc[c_idx_R] #right function coarse value

        #Fine Info
        f_idx_L = EToVf[e_id, 1] #Left point of fine element
        f_idx_R = EToVf[e_id, 2] #Right point of fine element

        #get fine point
        f_xL = xf[f_idx_L] #get left fine point
        f_xR = xf[f_idx_R] #get right fine point
        h_fine = f_xR - f_xL

        #Get fine function values
        f_uL = uhf[f_idx_L]
        f_uR = uhf[f_idx_R]
        #Interpolation

        #find r points
        r_L = (f_xL - c_xL) / h_coarse #r left point
        r_R = (f_xR - c_xL) / h_coarse #r right point


        #interpolation for coarse poin ts
        u_coarse_L = c_uL*(1-r_L) + c_uR*r_L
        u_coarse_R = c_uL*(1-r_R) + c_uR*r_R

        #difference between fine interpolation and coarse interpolation
        a0 = f_uL - u_coarse_L
        ah = f_uR - u_coarse_R

        #integration step and store result
        err[c_e_id] += L2_norm(a0,ah,h_fine)
    end

    return sqrt.(err)
end
```

c) Write a Matlab routine which can subdivide a coarse mesh and produce a refined mesh by refining only elements which has been marked for refinement (Same as in Exercise 1.6 b)).

We note that this is basically 1.6.b, but this time we add a Old2New list which keeps track of the old elements we changed.

```
#exercise1.7.c - Basically 1.6.b but we now keep track of the old elements we changed
function refine_marked(EToVcoarse, xcoarse, idxMarked)
    #Find number of marks
    idxMarked_set = Set(idxMarked) #lookup is O(1) so faster later
    new_addtions = length(idxMarked_set) #every new refined point "xm" will yield a new linep

    #--- Setup VX Table ---
    n_vx_old = length(xcoarse) #No of points.


    #Number of new VX elements
    # vx_new_elems = vx_old_elems + no_of_new_elems
    vx_points_new = n_vx_old + new_addtions

    #Instantiation
    vx_new = zeros(Float64, vx_points_new)
    vx_new[1:n_vx_old] = xcoarse #set all old points


    #--- Setup EToV ---
    n_elem_old = size(EToVcoarse,1) #Number of elements
    EToV_elements_new = n_elem_old + new_addtions
    EToV_new = zeros(Int, EToV_elements_new,2) #we need the connect from the left side and ri

    #-- Setup Etc. ---
    vx_cnt = n_vx_old + 1 #fill in points at the end of the old points
    elem_cnt = 1  #fill elements from beginning

    Old2New = zeros(Int, EToV_elements_new, 1) #tracker for old elements changed


    #We are going to iterate through all the elements and look to see which needs refinement!
    for elem_id in 1:1:n_elem_old

        #find points connected to element
        i_left = EToVcoarse[elem_id, 1] #"ei" left connection to point
        i_right = EToVcoarse[elem_id, 2] #"ei" right connection to point

        if elem_id in idxMarked_set

            #get points connected to element
            xi = xcoarse[i_left]  #left point
            xip1 = xcoarse[i_right] #right point

            #Add new elemnt to VX
            xm = (xi+xip1) / 2.0 #new x point
            vx_new[vx_cnt] = xm #add to new point "xm" VX table

            i_mid = vx_cnt #new point in VX

            #Add new "em" element connections to EToV
            EToV_new[elem_cnt, 1] = i_left #set left side of "em" to the previous left connec
            EToV_new[elem_cnt, 2] = i_mid #set the right side of "em" to the new point "xm"

            #Track "em" change
            Old2New[elem_cnt] = elem_id #we keep track of the old index of the old "e_i" elem
            elem_cnt += 1 #next row in EToV


            #Readjust old "ei" to be next element in EToV
            #Note that we need to reset all point connections to old "ei" because it has been
            EToV_new[elem_cnt, 1] = i_mid #set "ei" left connection to "xm"
            EToV_new[elem_cnt, 2] = i_right #set "ei" to the right connection
            #Track "old "ei"
            Old2New[elem_cnt] = elem_id

            elem_cnt += 1 #next row in EToV
            vx_cnt += 1 #next row in VX table
        else
            #Keep The element unrefned
            EToV_new[elem_cnt,1] = i_left #element remains connected to same left point
            EToV_new[elem_cnt,2] = i_right #element remains connceted to same right point
            #Notice that we do not move the VX row because this will be the next refinement p

            #Track no changes
            Old2New[elem_cnt] = elem_id

            elem_cnt += 1
        end

    end

    return vx_new, EToV_new, Old2New
end
```

d) Derive the weak formulation of the boundary value problem in the usual way and identify elemental contributions to A and $\mathbf{b}$ in the linear system that results from the discretization. It is allowed to assume that $f(x)$ can be represented in terms of a continuous piecewise polynomial function (see (1.2)). By this approximation the contributions to a right hand side vector $\mathbf{b}$ can be based on identifying the elemental contributions from

$$\int_0^L f(x)v(x)dx \approx \int_0^L \left(\sum_{j=1}^{M} \hat{f}_j N_j(x)\right) N_i(x)dx = \sum_{j=1}^{M} \hat{f}_j \int_0^L N_i(x)N_j(x)dx$$

Multiplying v on both sides, integrating from $[0, L]$

$$\int_0^L (u'' - u)v\ dx = \int_0^L vf(x)\ dx$$
$$\int_0^L u''v\ dx - \int_0^L uv\ dx = \int_0^L vf(x)\ dx$$

Note that $v(0) = v(L) = 0$, now using integration by parts $\int_0^L qp'\ dx = [uv]_0^L - \int_0^L q'p\ dx$, let $p = u'$ and $q = v$ then we can evaluate the first integral above.

$$\int_0^L u''v\ dx = [u'v]_0^L - \int_0^L v'u'\ dx = -\int_0^L v'u'\ dx$$
$$\Rightarrow \int_0^L u''v\ dx - \int_0^L uv\ dx = \int_0^L vf(x)\ dx$$
$$-\int_0^L v'u'\ dx - \int_0^L uv\ dx = \int_0^L vf(x)\ dx$$
$$-(\int_0^L v'u'\ dx + \int_0^L uv\ dx)) = \int_0^L vf(x)\ dx$$
$$(\int_0^L v'u'\ dx + \int_0^L uv\ dx)) = -\int_0^L vf(x)\ dx$$
$$\int_0^L (v'u' + uv)\ dx = -\int_0^L vf(x)\ dx$$

Using the hint we have

$$b = -\int_0^L vf(x)\ dx = -\sum_{j=1}^{M} \hat{f}_j \int_0^L N_i(x)N_j(x)\ dx$$

Consider $f(x) \approx \sum_{j=1}^{M} \hat{f}_j(x)N_j(x)$, then N_j is the global function. Then only two basis functions will contribute to this sum, see figure1.5, so we can write $\sum_{j=1}^{M} \hat{f}_j(x)N_j(x) = \hat{f}_j N_j(x) + \hat{f}_{j+1}N_{j+1}$ or in matrix form

$$\sum_{j=1}^{M} \hat{f}_j(x) = \begin{bmatrix} \hat{f}_j \\ \hat{f}_{j+1} \end{bmatrix}$$

Note that eq1.26 second term inside the integral is this exact expression

$$k_{r,s}^{(i)} = \int_{x_i}^{x_{i+1}} N_r^{(i)} N_s^{(i)}\ dx$$

Because of symmetry, it suffices to find $k_{1,1}^{(i)}, k_{1,2}^{(i)}$. Lastly We note that this is also half the integral of 1.1.a k-matrix. So we can do a variable substitution where $y = \frac{x-x_i}{h_i}$, hence

$$\begin{aligned} k_{1,1}^{(i)} &= \int_0^L N_1(y)N_1(y)\ dx = h\int_0^1 N_1(y)N_1(y)\ dy = -\frac{h}{3}[(1-y)^3]_0^1 = \frac{h}{3} \\ k_{1,2}^{(i)} &= \int_0^L N_1(x)N_2(x)\ dx = h\int_0^1 (1-y)(y)\ dy = h[\frac{1}{2}[y^2]_0^1 - \frac{1}{3}[y^3]_0^1] = \frac{h}{6} \end{aligned}$$

So we can write

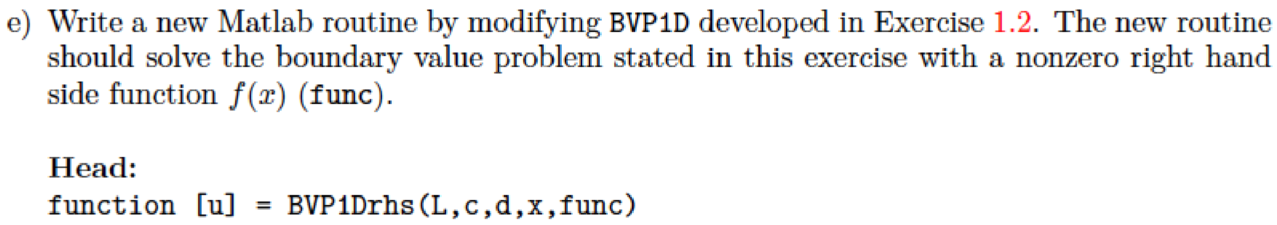
$$\begin{aligned} b &= -\sum_{j=1}^{M} \hat{f}_j \begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} = -\begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} \begin{bmatrix} \hat{f}_j \\ \hat{f}_{j+1} \end{bmatrix} \\ &= -\frac{h}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \hat{f}_j \\ \hat{f}_{j+1} \end{bmatrix} \end{aligned}$$

Using the results of 1.1.a we know the K-matrix to be

$$\kappa^{(\mathbf{i})} = \begin{bmatrix} k_{1,1}^{(i)} & k_{1,2}^{(i)} \\ k_{2,1}^{(i)} & k_{2,2}^{(i))} \end{bmatrix} = \begin{bmatrix} \frac{1}{h} + \frac{h_2}{3} & -\frac{1}{h_1} + \frac{h_1}{6} \\ -\frac{1}{h_1} + \frac{h_1}{6} & \frac{1}{h} + \frac{h_2}{3} \end{bmatrix}$$

Let $A_{sparse,\kappa}$ be similar in form as the A matrix with the $\kappa^{(i)}$ matrix found above. Then we can write up the final expression for the differential equation $u'' - u = f$ as

$$Au = b \Leftrightarrow A_{sparse,\kappa} \begin{bmatrix} \hat{u}_1 \\ \vdots \\ \hat{u}_n \end{bmatrix} = -\frac{h}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \hat{f}_j \\ \hat{f}_{j+1} \end{bmatrix}$$



e) Write a new Matlab routine by modifying `BVP1D` developed in Exercise 1.2. The new routine should solve the boundary value problem stated in this exercise with a nonzero right hand side function $f(x)$ (`func`).

Head:
`function [u] = BVP1Drhs(L,c,d,x,func)`

As requested in julia here is the code with the modified $BVP1D$ from exericse 1.2. Additionally we use the relevant implementations analytical solution of 1.7.a, the implementation of 1.7.b and 1.7.c and the analytical RHS found in 1.7.D

```
#Exercise1.7.e
using LinearAlgebra
using Plots

function assembly(M,h, fval)
    A = zeros(M,M)
    b = zeros(M)
    for i = 1:1:M-1
        k1_1 = 1/h[i] + h[i]/3
        k1_2 = -1/h[i] + h[i]/6
        k2_2 = 1/h[i] + h[i]/3

        A[i,i]      += k1_1
        A[i,i+1]    += k1_2
        A[i+1,i+1]  += k2_2

        #RHS dotproduct from final analytical RHS
        b1 = -(h[i]/6) * (2*fval[i] + 1*fval[i+1])
        b2 = -(h[i]/6) * (1*fval[i] + 2*fval[i+1])
        b[i]   += b1
        b[i+1] += b2
    end
    return A,b
end


function boundary_conditions(A,b,M,c,d)
    #A matrix
    A[1,1] = 1
    A[M,M] = 1

    #Left boundary
    b[1] = c
    b[2] = b[2]-A[1,2]*c
    A[1,2] = 0

    #right boundary
    b[M] = d
    b[M-1] = b[M-1] - A[M-1,M]*d
    A[M-1,M] = 0
    return A,b
```

```
end

function BVP1Drhs(L,c,d,M,func, u_exact)
    x = range(0,L, length=M) #equidistant
    h = diff(x)
    fx = func.(x)


    A = zeros(M,M)
    b = zeros(M,1)
    A,b = assembly(M,h, fx)
    A,b = boundary_conditions(A,b,M,c,d)

    # SOLVE SYSTEM
    chol = cholesky(Symmetric(A, :U), check=false)
    if issuccess(chol)
        u = chol \ b
    else
        println("A is not positive definite")
        return
    end

    # %% OUTPUT
    p = plot(x, u, label="FEM Solution", title="BVP 1D Solution Nonhomgenous", xlabel="x", yl
    plot!(p, x, u_exact.(x), label="exact u solution", linestyle=:dash)
    savefig(p, "w1/w1_7_e_FEM_exact.png")
    return x, u
end

#Setup
L = 1.0
M = 50
u_exact = x -> exp(-800*(x-0.4)^2) + 0.25*exp(-40*(x-0.8)^2)
c = u_exact(0)
d = u_exact(1)

#This is the exact solution for u''-u = f(x)
function f_RHS(x)
    term1 = (1600 * x^2 - 2560 * x + 1003.75) * exp(-1.6 * (5 * x - 4)^2)
    term2 = (2.56e6 * x^2 - 2.048e6 * x + 4.07999e5) * exp(-32 * (5 * x - 2)^2)
    return term1 + term2
end

x, u_approx = BVP1Drhs(L,c,d,M, f_RHS, u_exact)
```

f) Present relevant plots of computed solutions, element size distributions for converged solutions and evaluate the performance of the FEM + AMR algorithm implemented (e.g. figures showing error vs. DOF and CPU time vs. DOF).

We first write the code to generate the solutions

```
#Exercise1.7.e
using LinearAlgebra
using Plots

function assembly(M,h, fval)
    A = zeros(M,M)
    b = zeros(M)
    for i = 1:1:M-1
        k1_1 = 1/h[i] + h[i]/3
        k1_2 = -1/h[i] + h[i]/6
        k2_2 = 1/h[i] + h[i]/3

        A[i,i]      += k1_1
        A[i,i+1]    += k1_2
        A[i+1,i+1]  += k2_2

        #RHS dotproduct from final analytical RHS
        b1 = -(h[i]/6) * (2*fval[i] + 1*fval[i+1])
        b2 = -(h[i]/6) * (1*fval[i] + 2*fval[i+1])
        b[i]   += b1
        b[i+1] += b2
    end
    return A,b
end


function boundary_conditions(A,b,M,c,d)
    #A matrix
    A[1,1] = 1
    A[M,M] = 1

    #Left boundary
    b[1] = c
    b[2] = b[2]-A[1,2]*c
    A[1,2] = 0

    #right boundary
    b[M] = d
    b[M-1] = b[M-1] - A[M-1,M]*d
    A[M-1,M] = 0
    return A,b
end

function BVP1Drhs(x,L,c,d,func)
    #Sort refine_marked
    p = sortperm(x)

    #Intitialize
    x = x[p]
    h = diff(x)
    fx = func.(x)
    M = length(x)

    #Assembly process
    A = zeros(M,M)
    b = zeros(M,1)
    A,b = assembly(M,h, fx)
    A,b = boundary_conditions(A,b,M,c,d)

    # SOLVE SYSTEM
    chol = cholesky(Symmetric(A, :U), check=false)
    if issuccess(chol)
        u_sorted = chol \ b
    else
        println("A is not positive definite")
        return
    end

    #I have to re-sort because I fucked up on the mesh grid and didn't sort them, IDK man.
    u = zeros(M)
    u[p] = u_sorted
    return x, u
end



function AMR(u, tol, VX, EToVc, max_iter, L,c,d,f_RHS)

    history_dofs = []
    history_err = []
    history_cpu = [] # New array for timing

    u_curr = u #coarse solution

    for iter in 1:1:max_iter

        #Generate the refined mesh
        VXf, EToVf, Old2Newf = refine_marked(EToVc, VX, 1:size(EToVc,1))
        _, u_ref = BVP1Drhs(VXf, L, c, d, f_RHS)

        #use 1.7.b to compute error
        err_vec = errorestimate(VX, VXf, u_curr, u_ref, EToVc, EToVf, Old2Newf)
        max_err = maximum(err_vec) #NB! this is NOT the 1 norm. We can do this because we fou

        #store results
        push!(history_dofs, length(VX))
        push!(history_err, max_err)

        if max_err < tol #the maximum is below the threshold
            println("Converged at iteration $iter")
            break
        end

        idxMarks = findall(elem -> elem > tol, err_vec) #I go HARD and I take all the elmeent
        VX, EToVc, _ = refine_marked(EToVc, VX, idxMarks)

        #added this for the CPU time
        t_elapsed = @elapsed begin
            _, u_curr = BVP1Drhs(VX, L, c, d, f_RHS)
        end
        push!(history_cpu, t_elapsed)
    end

    return VX, EToVc, history_dofs, history_err,history_cpu #return the refined points and el
end
```

Now the code used for initializing we have

```
#Setup
#Inititalize EToV
M = 50 #first coarse grid
EToV = zeros(Int, M-1, 2)
for i in 1:M-1
    EToV[i, 1] = i      # Left point in element
    EToV[i, 2] = i + 1  # Right point in elmeent
end

L = 1.0
u_exact = x -> exp(-800*(x-0.4)^2) + 0.25*exp(-40*(x-0.8)^2)
c = u_exact(0)
d = u_exact(1)
VX = collect(range(0, L, length=M))

#find initial solution to feed into
_, u_init = BVP1Drhs(VX, L, c, d, f_RHS)


#Run AMR
tol = 1e-5
max_iter = 15
println("Starting AMR...")

VX_final, EToV_final, history_dofs, history_err, history_cpu = AMR(u_init, tol, VX, EToV, max
println("AMR Finished with $(length(VX_final)) points.")
```

And lastly we have the plot code

```
function required_plots(h_dofs, h_err,h_cpu)
    # Plot 1: Convergence (Error vs DoF)
    p1 = plot(h_dofs, h_err,
        xaxis=:log, yaxis=:log,
        marker=:circle,
        title="Convergence",
        xlabel="DoF (N)",
        ylabel="Max Error",
        label="Error",
        legend=:bottomleft)

    # Plot 2: CPU Time (Time vs DoF)
    p2 = plot(h_dofs[1:length(h_cpu)], h_cpu,
        xaxis=:log, yaxis=:log,
        marker=:square,
        color=:green,
        title="Computational Cost",
        xlabel="DoF (N)",
        ylabel="Solver Time (s)",
        label="CPU Time")

    # Plot 3: Mesh Adaptation
    x_plot = range(0, 1, length=1000)
    p3 = plot(x_plot, u_exact.(x_plot),
        line=:dash, color=:grey, label="Exact", title="Final Mesh")
    scatter!(p3, VX_final, u_exact.(VX_final),
        markersize=2, color=:red, label="Grid Nodes")

    # Combine all 3 plots
    final_plot = plot(p1, p2, p3, layout=(1,3), size=(1200, 400))
    display(final_plot)
    savefig(final_plot, "AMR_Full_Analysis.png")
end

required_plots(history_dofs, history_err, history_cpu)
```

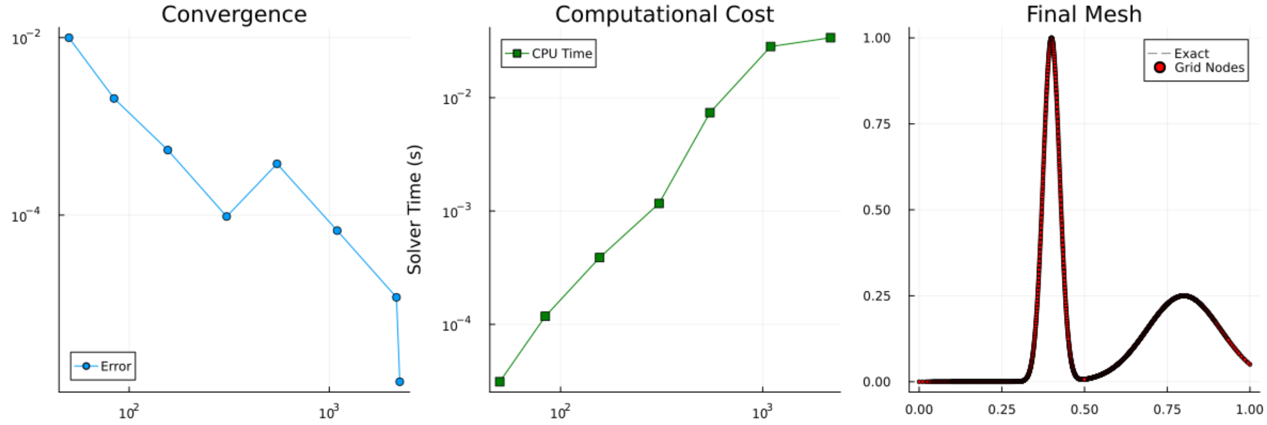


Figure 7: Three plots to analysize the function implemented in 1.7.e

We use tol $= 1e^{-5}$, max_iter $= 9$ and M=69 we generate the plots above this paragraph
Convergence We see a down trending, next week we could analyze the slope between neighbor points and expect to see a slope of -2 in the log log domain as the mesh increases.

Computation Cost As the mesh grid increases so does the computational power, according to the TA we would expect a linear curve this looks reasonable. Since we are using linear polynomial approximations, this is what we would expect for of the compute cost.

Final Mesh in red is plotted the exact solution using the analytical solution. And the dots are the approximations, visually these two are close to identical.

6 Appendix - Summary of Ai usage

Finite Element Method and AMR Implementation Log

Session Goal

Implementation of an Adaptive Mesh Refinement (AMR) loop for a 1D Boundary Value Problem using the Finite Element Method in Julia.

Part 1: Initial Code Analysis

User: I am looking at exercise 7.f, I want to do an AMR similar to 1.6.d, where is my mistake?
Code snippet provided: Initial implementations of `errorestimate`*,* `refine_marked`*, and* `BVP1Drhs`*.*
AI Analysis:

1. **Solver Logic:** The solver `BVP1Drhs` was generating a uniform mesh (`range`) internally, ignoring the refined mesh passed to it. It must instead accept `VX` and `EToV` as input arguments.
2. **AMR Loop:** The adaptive loop was missing a re-solve step. After refinement, the solution `u` becomes stale and must be recomputed on the new mesh before performing the next error estimation.
3. **Error Estimator:** The function `errorestimate` requires a reference ("fine") solution for comparison. The AMR loop did not include generation of this reference mesh and solution.
4. **Return Values:** The function `refine_marked` returns three values, but the calling code only unpacked two.

Part 2: Testing and Visualization

User: Let's make a plot to test this function.
AI Output: A Julia script was provided to visualize:

1. **Convergence:** A log–log plot of error versus degrees of freedom (DoF).
2. **Adaptivity:** A plot of the exact solution with grid nodes overlaid to verify mesh density in regions with high gradients.

Part 3: Debugging Variable Scope

User: This doesn't seem to work: `VX_final...  = AMR(u=u_init, ...)`
AI Analysis: The function call incorrectly used keyword arguments (e.g. `L=L`), while the function definition expected positional arguments. Additionally, a scoping issue was identified where `EToV_ref` was never defined, leading to an `UndefVarError`.
Correction: The `AMR` function signature was updated to explicitly accept problem parameters (`L`, `c`, `d`, `f_RHS`) rather than relying on global variables.

Part 4: Fixing the Topology Mismatch

User: *Code snippet provided with updated AMR call.*
AI Analysis: A logical error was identified in the refinement step, where the refined element connectivity was incorrectly constructed:

```
VX, EToVc, _ = refine_marked(EToVf, VX, idxMarks)
```