FINITE ELEMENT METHOD IN 1D – PART 2

In the previous lab we learned how to solve boundary value problems without any time dependence. Such problems are said to be stationary or steady, and they represent physical processes that have reached equilibrium.

In this lab we extend the methods we have learned to solve time-dependent (also called unsteady) problems. Specifically, we will consider the heat equation and the wave equation. Although we only consider problems in one spatial dimension here, the methods apply just as well to problems in higher dimensions.

1. The heat equation

The first problem we consider is the heat equation, which could be used to describe a diffusion process or the evolution of the temperature in a semi-infinite plate or a thin rod. For this problem the unknown function $u = u(x,t)$ depends on both the the spatial coordinate x and the time t. We seek a solution to the following problem,

$$u_t - (cu_x)_x = f \quad \text{in} \quad (a,b) \times (0,T], \tag{1a}$$

where subscripts u_t and u_x denotes partial derivatives with respect to t and x. As before, c and f are given functions (depending on both x and t), and we have a set of boundary conditions at the endpoints of the interval,

$$\left.\begin{aligned} u(a,t) &= u_a(t) \\ c(b,t)u_x(b,t) &= q_b(t) \end{aligned}\right\} \quad \text{for} \quad t \in (0,T]. \tag{1b}$$

Here the boundary conditions may also depend on the time t. For the heat equation we also need an *initial condition*,

$$u(x,0) = g(x) \quad \text{for} \quad x \in (a,b) \tag{1c}$$

Again, the first step in solving this problem is to derive a variational formulation. We multiply the equation in (1a) with a smooth test function v with $v(a) = 0$ (because there is a Dirichlet boundary condition at the left boundary) and integrate over the spatial domain (a,b), and we get

$$\int_a^b \big(u_t - (cu_x)_x\big)v\,dx = \int_a^b fv\,dx.$$

Now we do integration by parts on the second term on left hand side. This is exactly the same as in the previous time-independent case, and we get the variational problem: Find u such that

$$\int_a^b u_t(x,t)v(x) + cu_x(x,t)v_x(x)\,dx = \int_a^b f(x,t)v(x)\,dx + q_b(t)v(b) \qquad \forall v, \text{ with } v(a) = 0. \tag{2}$$

Here we have integrated away the dependence on the spatial coordinate x, but not on the time t, so (2) must hold hold for each $0 < t \leq T$.

2. A finite element method for the heat equation

The variational formulation (2) is the starting point for deriving a finite element formulation. In the previous lab, we discretised the variational form by partitioning the interval and introducing a subordinate basis of "hat functions" to represent the solution. In effect, the continuous coordinate x was replaced with a discrete set $x_1, x_2, \ldots, x_n$. For the current time-dependent problem, we will also have to similarly discretise the time coordinate t.

2.1. Discretising in space.

As for the stationary problem in the previous lab, we introduce a partitioning of the the interval (a, b), with $a = x_1 < x_2 < \ldots < x_n = b$, and we define a finite-dimensional space V_h, consisting of the continuous and piecewise linear functions:

$$V_h = \left\{ v \in C([a,b]) : v|_{[x_i, x_{i+1}]} \in \mathcal{P}_1,\ i = 1, \ldots, n-1 \right\}.$$

with the same basis $\{\varphi_j\}_{j=1}^n$ as before,

$$\varphi_j(x_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j. \end{cases}$$

Note that the functions in V_h vary in space, but not in time. We introduce the time dependence in the expansion coefficients with respect to the basis.

$$u_h(x,t) = \sum_{j=1}^{n} \xi_j(t)\varphi_j(x).$$

Now, in (2), we replace u with u_h and v with a basis function φ_i. We then have

$$\sum_{j=1}^{n} \int_a^b \dot{\xi}_j(t)\varphi_j(x)\varphi_i(x)\,dx + \sum_{j=1}^{n} \int_a^b c(t,x)\xi_j(t)\varphi_j'(x)\varphi_i'(x)\,dx$$
$$= \int_a^b f(x,t)\varphi_i(x)\,dx + q_b(t)\varphi_i(b) \qquad i = 1, 2, \ldots, n.$$

This is a system of first order ordinary differential equations for $\boldsymbol{\xi} = (\xi_1(t), \ldots, \xi_n(t))$. We rewrite it in a simpler form

$$\boldsymbol{M}\dot{\boldsymbol{\xi}}(t) + \boldsymbol{A}(t)\boldsymbol{\xi}(t) = \boldsymbol{b}(t) \tag{3}$$

where $\boldsymbol{M}$ is the *mass matrix*, and $\boldsymbol{A}$ and $\boldsymbol{b}$ are the stiffness matrix and loadvector as before, but now being time-dependent. The entries in $\boldsymbol{M}$, $\boldsymbol{A}$ and $\boldsymbol{b}$ are

$$m_{i,j} = \int_a^b \varphi_j(x)\varphi_i(x)\,dx$$
$$a_{i,j} = \int_a^b c(t,x)\xi_j(t)\varphi_j'(x)\varphi_i'(x)\,dx$$
$$b_i = \int_a^b f(x,t)\varphi_i(x)\,dx + q_b(t)\varphi_i(b).$$

Note that here we have not taken into account the effects of Dirichlet boundary conditions on the matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$. The stiffness matrix $\boldsymbol{A}$ and the load vector $\boldsymbol{b}$ are computed as for the stationary problem, but are now time-dependent.

2.2. **Using a diagonal mass matrix.** We will replace the exact mass matrix with an approximation, again making use of the trapezoidal rule to compute the integral. Recall that the basis functions have the property

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

The nonzero entries in the approximate mass matrix $\boldsymbol{M}$ are then the diagonal entries

$$m_{i,i} = \begin{cases} h/2 & \text{if } i = 1 \text{ or } i = n \\ h & \text{if } 1 < i < n. \end{cases}$$

2.3. **Discretising in time.** The semi-discrete system (3) have to be discretised in time. Suppose that $0 = t_1 < t_2 < \cdots < t_m = T$ is a partitioning of the time domain, and let $\boldsymbol{\xi}^l = \boldsymbol{\xi}(t_l)$, $l = 1, \ldots, m$. We replace the time derivative $\dot{\boldsymbol{\xi}}$ in (3) with an approximation

$$\frac{\boldsymbol{\xi}^{l+1} - \boldsymbol{\xi}^k}{t^{l+1} - t^l} \approx \dot{\boldsymbol{\xi}}(\theta t_{l+1} + (1-\theta)t_l), \qquad \theta \in [0, 1].$$

Usually, the parameter θ is 1 (backward differene), 0 (forward difference) or $1/2$ (central difference). Inserting this approximation into (3), using the notation $t_{l+\theta} = \theta t_{l+1} + (1-\theta)t_l$ we get,

$$\begin{aligned} \boldsymbol{M}\left(\frac{\boldsymbol{\xi}_{l+1} - \boldsymbol{\xi}_l}{t_{l+1} - t_l}\right) &\approx \boldsymbol{b}(t_{l+\theta}) - \boldsymbol{A}(t_{l+\theta})\boldsymbol{\xi}(t_{l+\theta}) \\ &\approx \boldsymbol{b}(t_{l+\theta}) - \theta\boldsymbol{A}(t_{l+\theta})\boldsymbol{\xi}(t_{l+1}) - (1-\theta)\boldsymbol{A}(t_{l+\theta})\boldsymbol{\xi}(t_l). \end{aligned}$$

For simplicty, we assume that the time steps are uniform, i.e. $t_{l+1} - t_l = k$, $l = 1, \ldots, m-1$, in which case $t_{l+\theta} = t_l + \theta k$. After rearranging the terms, we get an equation we can solve for $\boldsymbol{\xi}_{l+1}$:

$$\big(\boldsymbol{M} + \theta k \boldsymbol{A}(t_{l+\theta})\big)\boldsymbol{\xi}^{l+1} = \big(\boldsymbol{M} - (1-\theta)k\boldsymbol{A}(t_{l+\theta})\big)\boldsymbol{\xi}^l + k\boldsymbol{b}(t_{l+\theta}), \qquad l = 1, \ldots, m-1. \tag{4}$$

That is, for each $l = 1, \ldots, m-1$, we solve a linear system

$$\tilde{\boldsymbol{A}}\boldsymbol{\xi}^{l+1} = \tilde{\boldsymbol{b}},$$

where

$$\begin{aligned} \tilde{\boldsymbol{A}} &= \boldsymbol{M} + \theta k \boldsymbol{A}(t_{l+\theta}) \\ \tilde{\boldsymbol{b}} &= \big(\boldsymbol{M} - (1-\theta)k\boldsymbol{A}(t_{l+\theta})\big)\boldsymbol{\xi}^l + k\boldsymbol{b}(t_{l+\theta}) \end{aligned}$$

We can solve (4) sequentially for each $\boldsymbol{\xi}^{l+1}$, starting from $\boldsymbol{\xi}^1$ determined by the initial condition $u(x, 0) = g(x)$.

Equation (4) is a *time-stepping scheme* for solving time-dependent problems that generalises some of the commonly used schemes. Taking $\theta = 0$ results in a forward (or explicit) Euler scheme, $\theta = 1$ corresponds to backwards (or implicit) Euler, and $\theta = 1/2$ corresponds to Crank-Nicolson.

2.4. **Handling Dirichlet boundary conditions.** In (4) the Neumann boundary conditions are accounted for in the vector $\boldsymbol{b}(t_{k+\theta})$, but we still need to impose the Dirichlet Boundary condition.

Recall that the full system that we have to solve for in each time step reads

$$\tilde{\boldsymbol{A}}\boldsymbol{\xi}^{l+1} = \tilde{\boldsymbol{b}},$$

where

$$\begin{aligned}\tilde{\boldsymbol{A}} &= \boldsymbol{M} + \theta k \boldsymbol{A}(t_{l+\theta}) \\ \tilde{\boldsymbol{b}} &= \big(\boldsymbol{M} - (1-\theta)k\boldsymbol{A}(t_{l+\theta})\big)\boldsymbol{\xi}^l + k\boldsymbol{b}(t_{l+\theta})\end{aligned}$$

We impose the Dirichlet boundary condition the same as we did for the stationary problem in the previous lab. That is, we set the first row in $\tilde{\boldsymbol{A}}$ to identity, and fix the value in the first entry of $\tilde{\boldsymbol{b}}$.

Note that this means that the Dirichlet and Neumann boundary conditions are handled at different times. The Neumann boundary condition contributes to the load vector $\boldsymbol{b}(t_l + \theta k)$, but the Dirichlet boundary condition is imposed on the full system at time t_{l+1}.

2.5. **Error analysis for the time-stepping scheme.** The error at the final time T can be estimated according to[1]

$$\|u(T) - u_h(T)\|_2 \leq C_1 h^2 + C_2 k^r, \tag{5}$$

where $r = 1$ for forward and backward Euler, and $r = 2$ for Crank-Nicolson. The two terms on the right (5) comes from spatial and temporal discretisation errors, respectively.

When we want to test our implementation of a time-stepping scheme with the method of manufactured solutions, it is often convenient to choose a problem where there is no spatial discretion error. This happens when the exact solution can be exactly represented in the finite element basis. For example, if the exact solution is a line, then we make no error when approximating it with a piecewise linear function.

2.6. **Using MATLAB's solvers.** In this section we look at some other time-stepping methods. There is a number of solvers MATLAB that can be applied to systems of ODEs of the form

$$\boldsymbol{M}\dot{\boldsymbol{\xi}} = \boldsymbol{F}(t, \boldsymbol{\xi}).$$

We need to rewrite (3) in this form, and find

$$\boldsymbol{M}\dot{\boldsymbol{\xi}} = \boldsymbol{b}(t) - \boldsymbol{A}(t)\xi = \boldsymbol{F}(t, \boldsymbol{\xi}).$$

The available solvers are `ode15s`, `ode23`, `ode23s`, and `ode45`. For this problem, the implicit solvers `ode15s` and `ode23s` are the most appropriate. These usage of these solvers are

```
>> [T, U] = ode15s(F, [0,t_final], y0, odeopts)
```

where the variables involved are

[1] This estimate holds in general for $\theta \in [1/2, 1]$, but for $\theta \in [0, 1/2]$ it is only valid when $k \ll h$. This relates to the instability explicit methods for small values of the Courant number ck/h^2. See e.g. https://en.wikipedia.org/wiki/Courant_number.

- `F` is a function `F(t, y)` corresponding to $\boldsymbol{F}$. It must return a column vector,
- `t_final` is the final time,
- `y0` is a column vector containing the initial values,
- `odeopts` is an object containing solver options,
- `T` is a vector containing a partitioning of the time intervals,
- `U` is a matrix containing the solutions for the times in `T`. The solution at the final time can be accessed with `U(end, :)`.

The mass matrix $\boldsymbol{M}$ have to be passed to solver through the `odeopts` object. This can be done with the command `odeopts = odeset("Mass", M)`. Other options like solver tolerances can also be set the same way, see `help odeset` for details.

3. The wave equation

The next problem we will look at is the wave equation. This is similar to heat equation, except for a second derivative in time:

$$u_{tt} - (cu_x)_x = f \quad \text{in} \quad (a,b) \times (0,T], \tag{6a}$$

where subscripts u_t and u_x denotes partial derivatives with respect to t and x. As before, c and f are given functions (depending on both x and t), and we have a set of boundary conditions at the endpoints of the interval,

$$\left.\begin{aligned} u(a,t) &= u_a(t) \\ c(b,t)u_x(b,t) &= q_b(t) \end{aligned}\right\} \quad \text{for} \quad t \in (0,T]. \tag{6b}$$

For the wave equation we also need *two* initial conditions:

$$\left.\begin{aligned} u(x,0) &= g(x) \\ u_t(x,0) &= h(x) \end{aligned}\right\} \quad \text{for} \quad x \in (a,b). \tag{6c}$$

3.1. Solving the wave equation.

We can derive the variational form of the wave equation in exactly the same way as we did for the heat equation, and we can also discretise it the same way. In the end we get a system of ODEs that reads

$$\boldsymbol{M}\ddot{\boldsymbol{\xi}}(t) + \boldsymbol{A}(t)\boldsymbol{\xi}(t) = \boldsymbol{b}(t), \tag{7}$$

where the only difference between (3) and (7) is the order of the time derivative.

One way to attack the system (7) is to reformulate is a first order system by introducing an auxiliary variable $\boldsymbol{\chi} = \dot{\boldsymbol{\xi}}$. The resulting first order system can be written

$$\begin{aligned} \dot{\boldsymbol{\xi}}(t) &= \boldsymbol{\chi}(t) \\ \boldsymbol{M}\dot{\boldsymbol{\chi}}(t) &= \boldsymbol{b}(t) - \boldsymbol{A}(t)\boldsymbol{\xi}(t). \end{aligned}$$

In matrix notation, we have

$$\begin{bmatrix} \boldsymbol{I} & 0 \\ 0 & \boldsymbol{M} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{\xi}} \\ \dot{\boldsymbol{\chi}} \end{bmatrix} = \begin{bmatrix} 0 & \boldsymbol{I} \\ -\boldsymbol{A} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\chi} \end{bmatrix} + \begin{bmatrix} 0 \\ \boldsymbol{b} \end{bmatrix} \tag{8}$$

and this is problem that can we solve with MATLAB's ode solvers, for example `ode45`.

Exercise 1. We consider the following problem.

$$u_t - u_{xx} = 3x\exp(-3t) \quad \text{in} \quad (0,1) \times (0,1], \tag{9a}$$

with the boundary conditions

$$\left.\begin{aligned} u(0,t) &= 0 \\ u_x(1,t) &= 1 - \exp(-3t) \end{aligned}\right\} \quad \text{for} \quad t \in (0,1]. \tag{9b}$$

And the initial condition

$$u(x,0) = 0 \quad \text{for} \quad x \in (0,1). \tag{9c}$$

(a) Write a function `M = MassMatrix(N)` that computes the mass matrix for a uniform partitioning of $(0,1)$ into N subintervals using the approximate mass matrix from Section 2.2. Make sure to use the function `spdiags` to construct the matrix.

Also modify the function `StiffnessMatrix` that you implemented in the previous lab so that it uses `spdiags` to construct the sparse matrix.

(b) Copy `HeatEquation.m` into your working directory, where you have the functions `StiffnessMatrix`, `LoadVector` and `MassMatrix`. Check that the class works by typing in the following:

```
>> Problem = HeatEquation(4, @(x, t) 1, @(x, t) 1);
>> Problem.SolveTheta(1, 4, 1);
```

It might be necessary to make adjustments to `StiffnessMatrix` and `LoadVector` to make sure that the matrices are generated with compatible sizes. That is, the function call `LoadVector(N, f)` should generate a column vector of length $N+1$.

To implement problem (9) which we solve in this exercise, we can type in the following:

```
>> c = @(x, t) 1;
>> f = @(x, t) 3 * x * exp(-3*t);
>> N = 4;
>> Problem = HeatEquation(N, c, f);
>> Problem.BC_Types(1) = 'D';
>> Problem.BC_Values{2} = @(t) (1-exp(-3*t));
```

Here the last two lines changes the left boundary condition to type Dirichlet, and sets the value for the right boundary condition (by default, the boundary conditions are homogeneous Neumann).

(c) The class `HeatEquation` has a member function `HeatEquation.SolveTheta` for solving the heat equation with the time-stepping scheme described in section 2.3. However, this function is incomplete. The input parameter θ is unused, and the function works as if $\theta = 1$. Complete the function so that it implements the time stepping scheme described by equation (4).

After you fix the the function `SolveTheta`, try plotting some of the solutions with $\theta = 1/2$. Make a figure with space on one axis and time on the other, for example with `mesh` or `surf`. For example, we can type in the following code to solve the problem:

```
>> T_final = 1.0;
>> num_timesteps = 10;
>> [U, T] = Problem.SolveTheta(T_final, num_timesteps, 0.5);
```

Here `T` will hold the time partioning and `U` will have the solution. We can now plot the solution as follows:

```
>> X = linspace(0, 1, Problem.N + 1);
>> mesh(X, T, U);
```

Hint: The comments in the function explains every step. The lines that you have to modify are marked with *%FIXME* in the comments.

(d) Determine experimentally the convergence rate r in (5), for $\theta = 0, 1/2$ and 1. Do the values you get agree with the theoretical values (1,2 and 1, respectively)? You can make use of the function `HeatEquation.PrintConvergenceTableTheta` to print the tables. This function computes errors at the final time, and may be used as follows:

```
>> theta = 0.5;
>> u_e = @(x) x * (1 - exp(-3 * T_final));
>> Problem.PrintConvergenceTableTheta(u_e, [2,6], theta);
```

where `u_e` is the exact solution at the final time.

The exact solution to (9) is

$$u(x,t) = x(1 - \exp(-3t)),$$

and since the solution is linear in x, only the time discretisation contribute to the error in (5). Since the error does not depend on the spatial discretisation, you can use a small value of N when computing errors and rates, for example $N = 4$.

Exercise 2. We consider the following heat equation.

$$u_t - \big((1 + x^2t^2)u_x\big)_x = x/(1 + x^2t^2) \quad \text{in} \quad (0,1) \times (0,1], \tag{10a}$$

with the boundary conditions

$$\left.\begin{aligned} c(0,t)u_x(0,t) &= t \\ c(1,t)u_x(1,t) &= t \end{aligned}\right\} \quad \text{for} \quad t \in (0,1]. \tag{10b}$$

And the initial condition

$$u(x,0) = 0 \quad \text{for} \quad x \in (0,1). \tag{10c}$$

We want to solve this problem with the built-in ODE solvers in MATLAB, as described in section 2.6. These solvers require that the problem is written in the form $\boldsymbol{M}\dot{\boldsymbol{u}} = \boldsymbol{F}(t, \boldsymbol{u})$, where $\boldsymbol{F}$ must be provided as a MATLAB function.

(a) Extend the class `HeatEquation` with a function `y = F(t, u)` that we can used in conjunction with `ode15s`, as described in Section 2.6. You may assume that the boundary conditions are of Neumann type. As a starting point, consider the following code.

```
function y = F(obj, t, u)
    % F defines the right hand side of ode system equation
    A = obj.stiffnessmatrix(t);
    b = obj.loadvector(t);
    y = ...
end
```

Write the function in the `methods` part of the class definition in `heatequation.m`.

(b) Write a functions that solves the heat equation, using the built-in solver `ode15s`. Recall that `ode15s` is used as follows

```
>> [T, U] = ode15s(F, [0, t_final], u0, odeopts)
```

where `F` is a function defining the right hand side of the equation, the second argument is the time interval to solve for, and `u0` is the (known) solution at time $t = 0$. The last argument, `odeopts` is an object with various solver options. For example, we want to make use of the mass matrix and a sparse Jacobian. We can this by creating the object `odeopts` as follows:

```
>> JPattern = spdiags(ones(n, 3), [-1,0,1], n,n)
>> odeopts = odeset("Mass", M, "JPattern", JPattern)
```

To use the built in solver to solve (10), we need to pass in the function F from (a). Also, we need to interpolate the initial values g (which is zero for this specific problem). For example, this could be done in a function as follows:

```
function [T, U] = SolveODE15s(obj, t_final)
    % Assemble the mass matrix
    M = obj.MassMatrix();

    % Interpolate initial values
    x = linspace(0, 1, obj.N+1);
    u0 = zeros(obj.N + 1, 1);
    for i=1:(obj.N+1)
        u0(i) = obj.g(x(i));
    end

    % Solve
    F = @obj.F;
    odeopts = ...
    [T, U] = ...
end
```

After you have implemented this function, test it on problem (10), and plot the result, for example using `mesh` or `surf` as in exercise 1.

Remember that the problem have to set up with correct boundary conditions first. For example,

```
c = @(x, t) 1 + x^2 * t^2;
f = @(x, t) x / (1 + x^2 * t^2);
Problem = HeatEquation(50, c, f);
Problem.Boundary_Values = {@(t) t, @(t) t}
```

(c) The exact solution to (10) is

$$u(x,t) = \arctan(xt).$$

After we have solved the problem with the function implemented in (b), the error at the final time can be estimated as follows:

```
>> [T, U] = Problem.SolveODE15s(1.0);
>> u_h = U(end, :)'; % Extract values at the final time
>> u_e = @(x) atan(x);
>> error = Errornorm(u_e, u_h)
```

Solve the problem with $N = 50$ using `ode15s` and note the error and the number of timesteps needed.

(d) Try solving the problem with Crank-Nicolson ($\theta = 1/2$) and backwards Euler ($\theta = 1$) to the same accuracy. You will have to try different number of timesteps (and using the same value for N) and find one that gives approximately the same error as you got with `ode15s` in (c). How many timesteps are needed?

Exercise 3. We consider the wave equation,

$$u_{tt} - u_{xx} = 0 \quad \text{in} \quad (0,1) \times (0,1], \tag{11a}$$

with boundary conditions

$$\left.\begin{aligned} u_x(0,t) &= \pi \cos(\pi t) \\ u_x(1,t) &= \pi \cos(\pi - \pi t) \end{aligned}\right\} \quad \text{for} \quad t \in (0,1]. \tag{11b}$$

and initial conditions

$$\left.\begin{aligned} u(x,0) &= \sin(\pi x) \\ u_t(x,0) &= -\pi \cos(\pi x) \end{aligned}\right\} \quad \text{for} \quad x \in (0,1). \tag{11c}$$

The exact solution is

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

Make new class `WaveEquation`, as a copy of `HeatEquation`. Make sure file name, class name and constructor names are matching. To solve the wave equation (11), we need to make the following modifications:

(1) Add a `property` to hold the second initial condition.

(2) Modify the the function `WaveEquation.F` for the ODE system (2). Recall that the ODE system reads

$$\begin{bmatrix} \boldsymbol{I} & 0 \\ 0 & \boldsymbol{M} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{\xi}} \\ \dot{\boldsymbol{\chi}} \end{bmatrix} = \begin{bmatrix} 0 & \boldsymbol{I} \\ -\boldsymbol{A} & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\chi} \end{bmatrix} + \begin{bmatrix} 0 \\ \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\chi} \\ \boldsymbol{b} - \boldsymbol{A}\boldsymbol{\xi} \end{bmatrix}$$

This means that the vector of unknowns is twice as long as for it was for the heat equation, containing the nodal values of both u_h and $\dot{u}_h$. It is useful to split the vector and work with the individual components. For example,

```
function y = F(obj, t, u)
    u1 = u(1:n); u2 = u(n+1:end);
    ...
    y1 = u2; y2 = b - A * u1;
    y = [y1; y2];
```

(3) Add a function `WaveEquation.SolveODE45` similar to the solve function for the heat equation. The difference is that there are now two initial conditions to interpolate, and that the "mass matrix" for the system is different. Also, use the built-in solver `ode45` instead of `ode15s`.

Solve (11), plot the solution and compute the error.

Appendix A. Classes in matlab

It is often convenient to structure our code to make use of classes. In Matlab, classes has a set of properties (variables) and a set of methods (functions). We can define a class to store the parameters we need to define the heat equation as follows.

```
classdef HeatEquation
    %HEATEQUATION This class managed the data structures
    %             used to solve the heat equation.

    properties
        % We always need to following parameters
        N   % mesh size (number of intervals)
        c   % diffusion coefficient (function)
        f   % source term (function)

        % Default to homogeneous Neumann boundaries
        BC_Types  = ["N", "N"];
        BC_Values = {@(t) 0, @(t) 0};

        % Default to zero initial condition
        g = @(x) 0*x;
    end

    methods
        % Constructor
        function obj = HeatEquation(N, c, f)
            obj.N = N;
            obj.c = c;
            obj.f = f;
        end
    end
end
```

Just like functions, classes have to placed in a file with the same name as the class, in this case `HeatEquation.m`. We can then instantiate the class and change the parameters, for example

```
>> Problem = HeatEquation(8, @(x,t) 1, @(x,t) 3*x*exp(-3*t));
>> Problem.BC_Types(1) = "D";
>> Problem.BC_Vaues(2) = @(t) 1 - exp(-3*t);
```

The class can be extended with additional methods. For example, we can add a method for assembling the time-dependent stiffness matrix by including the following code under `methods` in the class definition:

```
        function [ A ] = StiffnessMatrix(obj, t)
        %STIFNESSSSMATRIX Assemble stiffness matrix at time t
            A = StiffnessMatrix( obj.N, @(x) obj.c(x,t) );
        end
```

Here we have defined a method `HeatEquation.StiffnessMatrix` that calls the function `StiffnessMatrix` that we implemented in the last lab. We can call this method to compute the stiffnessmatrix at time $t = 0.1$ by typing in

```
>> A = Problem = Problem.StiffnessMatrix(0.1);
```

This demonstrates some of the advantages of using classes. We do not have to pass around the all the variables, since the class already knows the variables N and c. Also, we do not have to create a new file for the new function, and avoid name conflicts. Classes can have local functions, meaning that we could copy the function `StiffnessMatrix` into `HeatEquation.m`, after the end of the class definition. This means that using classes is a way to get around MATLAB's limitation of one function per file.

A method to assemble the load vector $\boldsymbol{b}$, that takes as input a time-dependent source $f(x, t)$ can be implemented in a similar way.

```
function [ b ] = LoadVector(obj, t)
%LOADVECTOR Assemble the load vector at time t
    b = LoadVector( obj.N, @(x) obj.f(x, t) );

    % Account for boundary conditions
    if obj.BC_Types(1) == "N"
        b(1) = b(1) - obj.BC_Values{1}(t);
    end
    if obj.BC_Types(2) == "N"
        b(end) = b(end) + obj.BC_Values{2}(t);
    end
end
```