# Final Project
# for the national master's course
# Numerical Partial Differential Equations spring 2006

Albert-Jan Yzelman* & Tijmen Collignon†

August 22, 2006

## 1 The problem

In this report we will describe the Method of Lines and use this to solve the time-dependent elliptic boundary value problem:

$$\text{find } u \in C^2(\overline{\Omega}) \text{ such that } \begin{cases} \frac{\partial}{\partial t}u(x,y,z,t) & = \operatorname{div} K\nabla u(x,y,z,t) + qu(x,y,z,t) \text{ on } \Omega \times [0,T] \\ u(x,y,z,t) & = 0 \text{ on } \partial\Omega \times [0,T] \\ u(x,y,z,0) & = u_0(x,y,z) \end{cases}$$

using the continuous piecewise quadratic finite element method. We also have $q \geq 0$ and $K$ is a user-specified symmetric positive definite $3 \times 3$ matrix. Writing the partial differential equation explicitly results in

$$\begin{aligned} \frac{\partial}{\partial t}u =& (k_{11}+k_{21}+k_{31})\frac{\partial^2}{\partial x^2}u \\ &+(k_{12}+k_{22}+k_{32})\frac{\partial^2}{\partial y^2}u \\ &+(k_{13}+k_{23}+k_{33})\frac{\partial^2}{\partial z^2}u \\ &+qu \qquad\qquad \text{on } \Omega \times [0,T]. \end{aligned}$$

## 2 Weak formulation

First, we define $u(t) = u(\cdot, t) \in H_0^1(\Omega)$. The following weak formulation is for each $t \in [0,T]$. We want to find a function $u(t) \in H_0^1(\Omega)$ such that

$$\begin{aligned} a(u(t), v) &= \left(\frac{\partial}{\partial t}u(t), v\right) \qquad \forall v \in H_0^1(\Omega) \\ u(0) &= u_0 \in H_0^1(\Omega) \end{aligned} \tag{1}$$

*Electronic address: ajy777@gmail.com
†Electronic address: tpcollig@math.uu.nl

where the bilinear form $a(\cdot,\cdot)$ is defined as

$$a(u(t),v) = -(K\nabla u(t), \nabla v) + q(u(t),v). \tag{2}$$

This is called the *weak formulation* of our boundary value problem and because the bilinear form (2) is coercive and continuous on $H_0^1(\Omega)\times H_0^1(\Omega)$ the Lax Milgram Theorem ensures us that this solution $u(t)$ exists and that it is unique.

# 3 Quadratic tetrahedral finite element method

To approximate the weak solution $u(t)$ of (1) we use the quadratic tetrahedral finite element method. We choose to use the space $V(\mathcal{T}_h)$ of continuous piecewise quadratic functions with respect to a given tetrahedralization $\mathcal{T}_h$ of $\Omega$. More specifically,

$$V(\mathcal{T}_h) = \{v \in C^0(\Omega) \mid v_{|H} \in \mathcal{P}^2(H) \text{ for all } H \in \mathcal{T}_h\}. \tag{3}$$

We are particularly interested in the space

$$V_0(\mathcal{T}_h) = V(\mathcal{T}_h) \cap H_0^1(\Omega). \tag{4}$$

## 3.1 Finite element formulation (semi-discrete formulation)

For each $t$, find $u_h(t) \in V_0(\mathcal{T}_h) \subset H_0^1(\Omega)$ such that

$$\begin{aligned} a(u_h(t), v_h) &= \left(\frac{\partial}{\partial t}u_h(t), v_h\right) \text{ for all } v_h \in V_0(\mathcal{T}_h) \\ u_h(0) &= \Pi_h u_0 \end{aligned} \tag{5}$$

where

$$\Pi : H_0^1(\Omega) \to V_0(\mathcal{T}_h).$$

The reason that we use $\Pi_h$ instead of something like $L_h f$ is that even if you solve equation (5) exactly at $t = 0$, this will generally not hold for $t > 0$.

We will now write $m = \dim(V_0(\mathcal{T}_h))$ and $m + l = \dim(V(\mathcal{T}_h))$. After choosing *any* basis $(\phi_j)_{j=1}^m$ for $V_0(\mathcal{T}_h)$ we have for each $t$

$$u_h(t) = \sum_{j=1}^{m} \alpha_j(t)\phi_j.$$

We also have

$$\frac{\partial}{\partial t}u_h(t) = \frac{\partial}{\partial t}\sum_{j=1}^{m}\alpha_j(t)\phi_j = \sum_{j=1}^{m}\alpha_j'(t)\phi_j$$

where the apostrophe denotes time derivative. Plugging this in (5) gives

$$a\left(\sum_{j=1}^{m}\alpha_j(t)\phi_j, \phi_i\right) = \left(\sum_{j=1}^{m}\alpha_j'(t)\phi_j, \phi_i\right) \text{ for all } \phi_i \in V_0(\mathcal{T}_h)$$

which is equal to the system

$$\begin{bmatrix} a(\phi_1,\phi_1) & \cdots & a(\phi_m,\phi_1) \\ \vdots & & \vdots \\ a(\phi_1,\phi_m) & \cdots & a(\phi_m,\phi_m) \end{bmatrix} \begin{bmatrix} \alpha_1(t) \\ \vdots \\ \alpha_m(t) \end{bmatrix} = \begin{bmatrix} (\frac{\partial}{\partial t}u(t),\phi_1) \\ \vdots \\ (\frac{\partial}{\partial t}u(t),\phi_m) \end{bmatrix} \equiv M \begin{bmatrix} \alpha_1'(t) \\ \vdots \\ \alpha_m'(t) \end{bmatrix}.$$

The system matrix of the left hand side can be written as a linear combination $A + qM$ and the righthand side can be written as $M\alpha'(t)$, where

$$A = -\begin{bmatrix} (K\nabla\phi_1,\nabla\phi_1) & \cdots & (K\nabla\phi_m,\nabla\phi_1) \\ \vdots & & \vdots \\ (K\nabla\phi_1,\nabla\phi_m) & \cdots & (K\nabla\phi_m,\nabla\phi_m) \end{bmatrix} \quad \text{and} \quad M = \begin{bmatrix} (\phi_1,\phi_1) & \cdots & (\phi_m,\phi_1) \\ \vdots & & \vdots \\ (\phi_1,\phi_m) & \cdots & (\phi_m,\phi_m) \end{bmatrix}$$

resulting in the system

$$(A + qM)\alpha(t) = M\alpha'(t) \tag{6}$$

where $\alpha(0)$ is determined by $\Pi_h$ and $\{\phi_j\}_j$.

This is a system of ordinary differential equations (ODEs) which we will solve using the one-step method Euler forward. So we have to solve for $\alpha$ (numerically) and then plug it into the finite element formulation. This amounts to two approximations.

# 4 Choice of basis

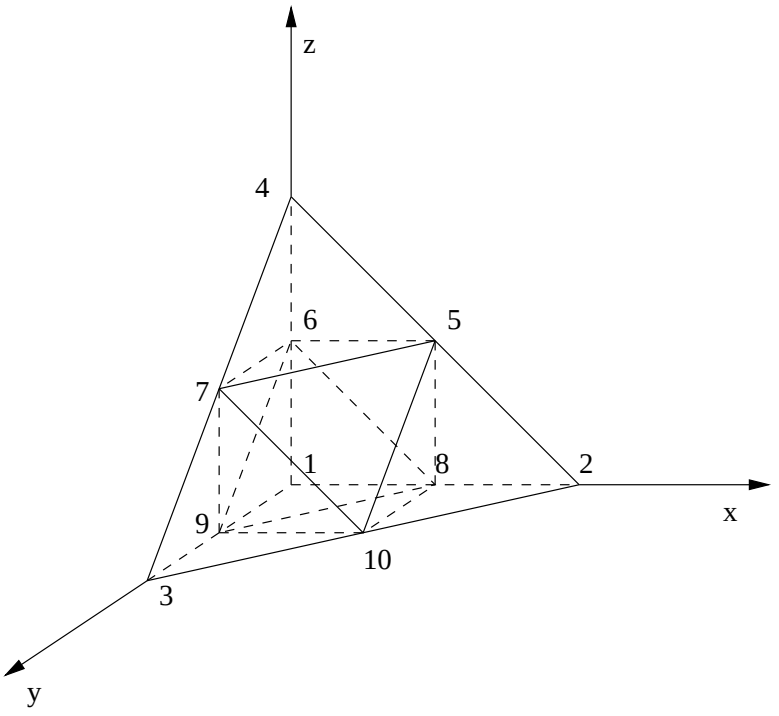



Figure 1: The unit tetrahedron

We are going to use the nodal basis functions $\{\phi_j\}_{j=1}^{m+l} \in V(\mathcal{T}_h)$ with nodes at the vertices and at the midpoints of the edges of a tetrahedron. We want to find a nodal basis $\{\phi_j\}_{j=0}^{m+l}$ that satisfies

$$\phi_j(e_i) = \delta_{ij}$$

where $\delta_{ij}$ is the Kronecker delta and the $e_i$ are the vertices and midpoints of the tetrahedra. A simple basis for the unit tetrahedron is given by

$$\begin{aligned}\psi_1 &= x,\\ \psi_2 &= y,\\ \psi_3 &= z,\\ \psi_4 &= 1 - x - y - z.\end{aligned}$$

Each nodal basis function on an arbitrary tetrahedron can be constructed by stretching and shifting the following ten functions

$$\begin{gathered}\Phi_1 = 2\psi_4(\psi_4 - p), \Phi_2 = 2\psi_1(\psi_1 - p), \Phi_3 = 2\psi_2(\psi_2 - p), \Phi_4 = 2\psi_3(\psi_3 - p)\\ \Phi_5 = 4\psi_1\psi_3, \Phi_6 = 4\psi_3\psi_4, \Phi_7 = 4\psi_2\psi_3\\ \Phi_8 = 4\psi_1\psi_4, \Phi_9 = 4\psi_2\psi_4, \Phi_{10} = 4\psi_1\psi_2\end{gathered}$$

which are evaluated at the vertices and at the midpoints $\nu_i$, which are

$$\nu_i \in \{(0,0,0), (1,0,0), (0,1,0), (0,0,1), (p,0,p), (0,0,p), (0,p,p), (p,0,0), (0,p,0), (p,p,0)\}$$

shown in Figure 1. Here, $p = \frac{1}{2}$. These functions are obtained by considering each node separately. For instance, looking at the seventh node $\nu_7 = (0,p,p)$, all the points that need to be zero lie on the $xz$-plane and on the $xy$-plane. Therefore, if we multiply the basis functions that are zero here (i.e. $\psi_2$ and $\psi_3$) and normalize, we obtain $\Phi_7 = 4\psi_2\psi_3$.

### 4.1 Transformation to an arbitrary tetrahedron

Let $H$ be an arbitrary tetrahedron with vertices $v^i = (x_i, y_i, z_i)^* \in \mathbb{R}^3$ for $i = 1,2,3,4$. We define the affine transformation from the unit tetrahedron to an arbitrary tetrahedron as

$$T_k^{-1} : \widehat{H} \to H_k : \begin{bmatrix}x\\y\\z\end{bmatrix} \mapsto \begin{bmatrix}x_1\\y_1\\z_1\end{bmatrix} + \begin{bmatrix}x_2 - x_1 & x_3 - x_1 & x_4 - x_1\\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1\\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1\end{bmatrix} \begin{bmatrix}x\\y\\z\end{bmatrix}. \tag{7}$$

Indeed we find that for instance $(0,0,0)^* \mapsto (x_1, y_1, z_1)^*$. Then we have that the ten basis functions on the reference tetrahedron transform according to

$$\phi_j = \Phi_j \circ T_k \qquad \text{for } j = 1, 2, \ldots, 10.$$

By using this expression we can compute integrals over arbitrary tetrahedra by using the substitution theorem for integrals.

### 4.2 Local mass and stiffness matrices

The so-called element stiffness and mass matrices are defined by

$$E_A := \begin{bmatrix} (K\nabla\Phi_1, \nabla\Phi_1) & \cdots & (K\nabla\Phi_{10}, \nabla\Phi_1) \\ \vdots & & \vdots \\ (K\nabla\Phi_{10}, \nabla\Phi_1) & \cdots & (K\nabla\Phi_{10}, \nabla\Phi_{10}) \end{bmatrix} \quad \text{and} \quad E_M := \begin{bmatrix} (\Phi_1, \Phi_1) & \cdots & (\Phi_{10}, \Phi_1) \\ \vdots & & \vdots \\ (\Phi_{10}, \Phi_1) & \cdots & (\Phi_{10}, \Phi_{10}) \end{bmatrix}.$$

We will now discuss how to construct these matrices.

**Element stiffness matrix.** We start by computing the gradients of the ten unit basis functions.

$$\nabla\Phi_1 = \begin{bmatrix} -3+4(x+y+z) \\ -3+4(x+y+z) \\ -3+4(x+y+z) \end{bmatrix}, \nabla\Phi_2 = \begin{bmatrix} 4x-1 \\ 0 \\ 0 \end{bmatrix}, \nabla\Phi_3 = \begin{bmatrix} 0 \\ 4y-1 \\ 0 \end{bmatrix},$$

$$\nabla\Phi_4 = \begin{bmatrix} 0 \\ 0 \\ 4z-1 \end{bmatrix}, \nabla\Phi_5 = \begin{bmatrix} 4z \\ 0 \\ 4x \end{bmatrix}, \nabla\Phi_6 = \begin{bmatrix} -4z \\ -4z \\ 4-4x-4y-8z \end{bmatrix},$$

$$\nabla\Phi_7 = \begin{bmatrix} 0 \\ 4z \\ 4y \end{bmatrix}, \nabla\Phi_8 = \begin{bmatrix} 4-8x-4y-4z \\ -4x \\ -4x \end{bmatrix}, \nabla\Phi_9 = \begin{bmatrix} -4y \\ 4-4x-8y-4z \\ -4y \end{bmatrix}, \nabla\Phi_{10} = \begin{bmatrix} 4y \\ 4x \\ 0 \end{bmatrix}.$$

As an example, if we take the second and third basis functions, we find

$$E_A^{3,2} = (K\nabla\Phi_3, \nabla\Phi_2) = \int_{\widehat{H}} (\nabla\Phi_3)^* K(\nabla\Phi_2)\ dxdydz = \int_{\widehat{H}} \begin{bmatrix} 0 \\ 4y-1 \\ 0 \end{bmatrix}^* K \begin{bmatrix} 4x-1 \\ 0 \\ 0 \end{bmatrix}\ dxdydz$$

$$= \int_{\widehat{H}} k_{21}(4y-1)(4x-1)\ dxdydz = \int_{\widehat{H}} k_{21}(1-4x-4y+16xy)\ dxdydz = -\frac{1}{30}k_{21}. \quad (8)$$

Instead of computing 55 different integrals, we will use the linearity of integration and the symmetry of the unit tetrahedron. First we compute the integrals

$$\int_{\widehat{H}} 1\ dxdydz = \frac{1}{6}, \qquad \int_{\widehat{H}} x\ dxdydz = \frac{1}{24}, \quad (9)$$

$$\int_{\widehat{H}} xy\ dxdydz = \frac{1}{120}, \qquad \int_{\widehat{H}} x^2\ dxdydz = \frac{1}{60} \quad (10)$$

which represent all the possible terms that we might encounter in the integrals. The symmetry of the tetrahedron ensures us that for instance $\int x^2\,dxdydz = \int y^2\,dxdydz$. We then construct the four coefficient matrices that belong to the gradients. For instance, $\Omega_{(y)}$ contains the

coefficients that belong to the $y$ terms in the gradients. Specifically, these matrices are

$$\Omega_{(1)} = \begin{bmatrix} -3 & -1 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ -3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ -3 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{11}$$

$$\Omega_{(x)} = \begin{bmatrix} 4 & 4 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & -4 & -4 & 4 \\ 4 & 0 & 0 & 0 & 4 & -4 & 0 & -4 & 0 & 0 \end{bmatrix}, \tag{12}$$

$$\Omega_{(y)} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 & 0 & -4 & -4 & 4 \\ 4 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & -8 & 0 \\ 4 & 0 & 0 & 0 & 0 & -4 & 4 & 0 & -4 & 0 \end{bmatrix}, \tag{13}$$

$$\Omega_{(z)} = \begin{bmatrix} 4 & 0 & 0 & 0 & 4 & -4 & 0 & -4 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & -4 & 4 & 0 & -4 & 0 \\ 4 & 0 & 0 & 4 & 0 & -8 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{14}$$

If we want to calculate the element stiffness matrix we have to consider the possible combinations of $x, y, z$, and the constant terms that results in the multiplication of the gradients. In other words, using the example given in (8), we get the expression

$$E_A^{3,2} = \left[ \frac{1}{6}\Omega_{(1)}^* K \Omega_{(1)} + \frac{1}{24}\left( \Omega_{(1)}^* K \Omega_{(x)} + \Omega_{(y)}^* K \Omega_{(1)} \right) + \frac{1}{120}\Omega_{(y)}^* \Omega_{(x)} \right]_{3,2} \tag{15}$$

$$= \left( \frac{1}{6} - \frac{4}{24} - \frac{4}{24} + \frac{16}{120} \right) k_{21} = -\frac{1}{30} k_{21}. \tag{16}$$

As for the general case, we have to include *every* possible combination of the $x, y, z$, and constant terms, resulting in the expression

$$\begin{aligned} E_A = {} & \frac{1}{6}\Omega_{(1)}^* K \Omega_{(1)} \\ & + \frac{1}{24}\left( \Omega_{(1)}^* K \Omega_{(x)} + \Omega_{(1)}^* K \Omega_{(y)} + \Omega_{(1)}^* K \Omega_{(z)} + \Omega_{(x)}^* K \Omega_{(1)} + \Omega_{(y)}^* K \Omega_{(1)} + \Omega_{(z)}^* K \Omega_{(1)} \right) \\ & + \frac{1}{120}\left( \Omega_{(x)}^* K \Omega_{(y)} + \Omega_{(y)}^* K \Omega_{(x)} + \Omega_{(x)}^* K \Omega_{(z)} + \Omega_{(z)}^* K \Omega_{(x)} + \Omega_{(y)}^* K \Omega_{(z)} + \Omega_{(z)}^* K \Omega_{(y)} \right) \\ & + \frac{1}{60}\left( \Omega_{(x)}^* K \Omega_{(x)} + \Omega_{(y)}^* K \Omega_{(y)} + \Omega_{(z)}^* K \Omega_{(z)} \right). \end{aligned} \tag{17}$$

We would like to transform the element stiffness matrix to an arbitrary tetrahedron. Using the transformation given in (7) and the chain rule we can write

$$\nabla\phi = \nabla(\Phi \circ T) = [D(\Phi \circ T)]^* = [(D\Phi \circ T)(DT)]^* = (DT)^*(D\Phi \circ T)^* = (DT)^*(\nabla\Phi \circ T)$$

where we have omitted the indices for clarity. Using the same basis functions as an example as before, we get for an arbitrary tetrahedron

$$\int_H (\nabla\phi_3)^* K (\nabla\phi_2)\ dxdydz = \int_H (\nabla\Phi_3 \circ T)^* (DT) K (DT)^* (\nabla\Phi_2 \circ T)\ dxdydz \tag{18}$$

$$= \int_{\widehat{H}} (\nabla\Phi_3)^* (DT) K (DT)^* (\nabla\Phi_2) |\det(DT)^{-1}|\ dxdydz \tag{19}$$

where we have used the Substitution Theorem in the second equality. We also know that

$$DT = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{bmatrix}^{-1}$$

so $DT$ is constant on $H$. The transformed element stiffness matrix can thus be easily obtained by replacing $K$ in (17) with

$$\frac{(DT)K(DT)^*}{|\det(DT)|}.$$

**Element mass matrix.** The integrals in this matrix can be evaluated very easily. For instance

$$E_M^{3,2} = (\Phi_3, \Phi_2) = \int_{\widehat{H}} \Phi_3\Phi_2 \; dxdydz = \int_0^1 \int_0^{1-x} \int_0^{1-x-y} 2y(y - \frac{1}{2})2x(x - \frac{1}{2}) \; dzdydx = \frac{1}{2520}.$$

Calculating the 55 different integrals gives the following symmetric matrix

$$\overline{E}_M := \begin{bmatrix} 420 & & & & & & & & & \\ 2520 & 420 & & & & & & & & \\ 2520 & 2520 & 420 & & & & & & & \\ 2520 & 2520 & 2520 & 420 & & & & & & \\ -420 & -630 & -420 & -630 & 315/4 & & & & & \\ -630 & -420 & -420 & -630 & 315/2 & 315/4 & & & & \\ -420 & -420 & -630 & -630 & 315/2 & 315/2 & 315/4 & & & \\ -630 & -630 & -420 & -420 & -420 & 315/2 & 315 & 315/4 & & \\ -630 & -420 & -630 & -420 & 315 & 315/2 & 315/2 & 315/2 & 315/4 & \\ -420 & -630 & -630 & -420 & 315/2 & 315 & 315/2 & 315/2 & 315/2 & 315/4 \end{bmatrix}$$

where we have to take the reciprocal of each element to obtain the element mass matrix $E_M$. Also, the upper triangular part is omitted to avoid clutter.

Transforming this to an arbitrary tetrahedron is done in a similar way as with the stiffness matrix:

$$\int_H \phi_3\phi_2 \; dxdydz = \int_H (\Phi_3 \circ T)(\Phi_2 \circ T) \; dxdydz = \int_{\widehat{H}} \Phi_3\Phi_2 |\det(DT)^{-1}| \; dxdydz. \tag{20}$$

In other words, the transformed element mass matrix can be written as

$$\frac{E_M}{|\det(DT)|}. \tag{21}$$

## 5 Assembling the global stiffness and mass matrices

To construct the global mass and stiffness matrix for $V(\mathcal{T}_h)$, we first obtain the $(m+l)\times(m+l)$ matrices $\hat{A}$ and $\hat{M}$ of the space $V(\mathcal{T}_h)$ and then remove the $l$ functions that belong to the vertices and midpoints that are on the edge. In order to do this, it can be shown that

$$a(\phi_i, \phi_j) = \sum_{k=1}^{n} \int_{H_k} (-[K\nabla\phi_i(x,y,z)]^*\nabla\phi_j(x,y,z) + q\phi_i(x,y,z)\phi_j(x,y,z)) \; dxdydz$$

where $n$ is the number of tetrahedra in the tetrahedralization. Using this, it is straightforward to show that

$$\widehat{A} = \sum_{k=1}^{n} \widehat{A}_k \quad \text{and} \quad \widehat{M} = \sum_{k=1}^{n} \widehat{M}_k \tag{22}$$

where

$$\widehat{A}_k = \left[ \int_{H_k} [K \nabla \phi_i(x,y,z)]^* \nabla \phi_j(x,y,z) \; dxdydz \right]_{ij} \tag{23}$$

$$\text{and} \quad \widehat{M}_k = \left[ \int_{H_k} \phi_i(x,y,z) \phi_j(x,y,z) \; dxdydz \right]_{ij}. \tag{24}$$

Note that on an arbitrary tetrahedron, only the ten nodal basis functions $\{\phi_j\}$ are non-zero.

## 6 ODE solvers

To solve the system of ordinary differential equations given in (6), we can use the Euler forward method. For the general case, we want to solve the system

$$\begin{cases} \alpha'(t) = f(t, \alpha(t)), & t \in [0,T] \\ \alpha(t_0) = \alpha_0 \end{cases}$$

for some $f$. In our particular case, this leads to the system

$$\begin{cases} \alpha'(t) = M^{-1}(A + qM)\alpha(t), & t \in [0,T] \\ \alpha(0) = \alpha_0 \end{cases}$$

where $\alpha_0$ is determined by the nodal basis functions $\{\phi_j\}$.

Applying the Euler forward method to this system results in the approximating sequence $(\widetilde{\alpha}_i)_{i \geq 0}$, where

$$\widetilde{\alpha}_{i+1} = \widetilde{\alpha}_i + k(M^{-1}A + qI)\widetilde{\alpha}_i = (I + k(M^{-1}A + qI))\widetilde{\alpha}_i \tag{25}$$

where $k$ is the stepsize. This method is of order $\mathcal{O}(k)$.

## 7 Testing the finite element part

To investigate whether the implementation of the finite element part and the refinement procedure is working correctly, we will study the convergence by using as an exact solution

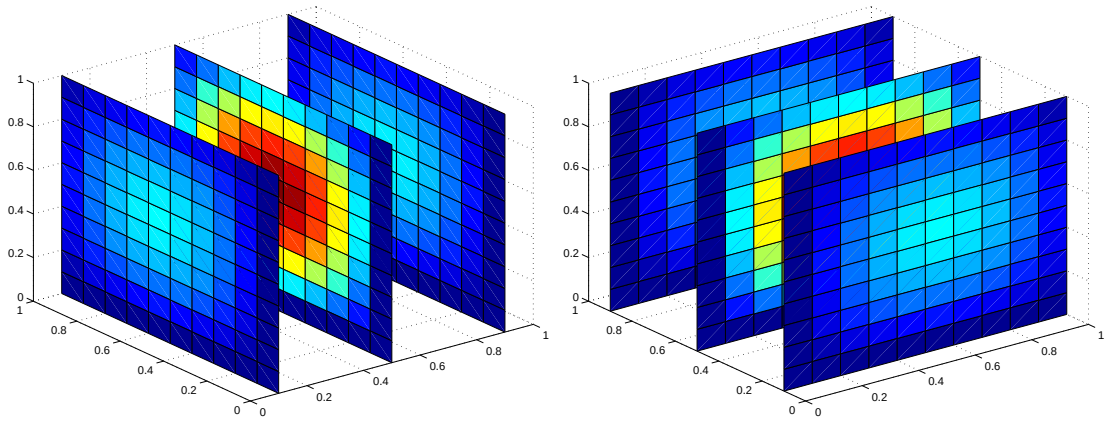
$$u(x,y,z) = xyz(1-x)(1-y)(1-z). \tag{26}$$

To do this, we have to compute the three second order partial derivatives

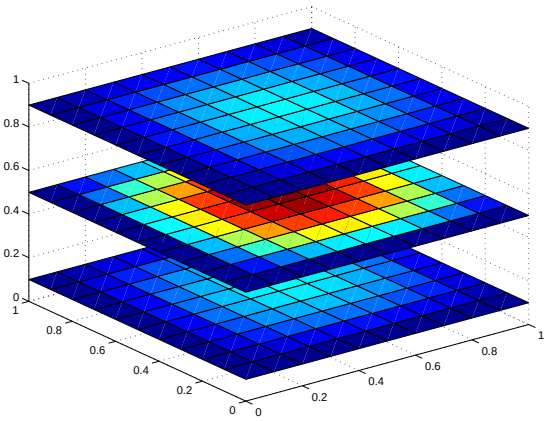
$$\frac{\partial^2}{\partial x^2} u = -2yz(y-1)(z-1), \tag{27}$$

$$\frac{\partial^2}{\partial y^2} u = -2xz(x-1)(z-1), \tag{28}$$

$$\frac{\partial^2}{\partial z^2} u = -2xy(x-1)(y-1). \tag{29}$$



(a) (b)



(c)

Figure 2: Slices of the function $u$

In other words, we will solve

$$\text{given } f \in C_0(\Omega), \text{ find } u \in C^2(\overline{\Omega}) \text{ such that } \begin{cases} \operatorname{div} K\nabla u + qu & = f \text{ on } \Omega \\ u & = 0 \text{ on } \partial\Omega \end{cases} \tag{30}$$

where we take $q = 1, K = I$, and as domain $\Omega$ the unit cube and

$$\begin{aligned} f(x,y,z) = &- 2yz(y-1)(z-1) - 2xz(x-1)(z-1) - 2xy(x-1)(y-1) \\ &+ xyz(1-x)(1-y)(1-z) \end{aligned}$$

and compare the approximated solution with the exact solution $u$. Shown in Figure 2 are slices of the function $u$ along the $x, y$, and $z$ directions to help visualize the function.

In this case, the finite element approximation $u_h$ is the unique element from $V_0(\mathcal{T}_h)$ which satisfies

$$a(u_h, v_h) = (f, v_h) \text{ for all } v_h \in V_0(\mathcal{T}_h). \tag{31}$$

To evaluate the integrals on the right-hand side, we propose to replace the function $f$ with its quadratic interpolant $L_h^2 f \in V(\mathcal{T}_h)$ on the vertices and midpoints of the tetrahedra. This allows us to solve the integrals exactly, although we will now be solving for $\widehat{u}_h$. That is, we obtain the $\widehat{u}_h$ from

$$a(\widehat{u}_h, v_h) = (L_h^2 f, v_h) \text{ for all } v_h \in V_0(\mathcal{T}_h) \tag{32}$$

where

$$L_h^2 f = \sum_{j=1}^{m+l} f(e_j)\phi_j. \tag{33}$$

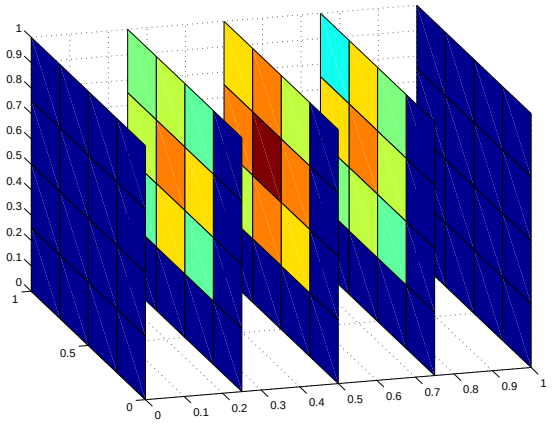
Figure 3 shows slices of the approximation $\widehat{u}_h$ using our implementation of the finite element method for two refinements. If we compare these to the exact solution given in Figure 2 we conclude that the finite element method is working correctly.

# 8 Conclusion

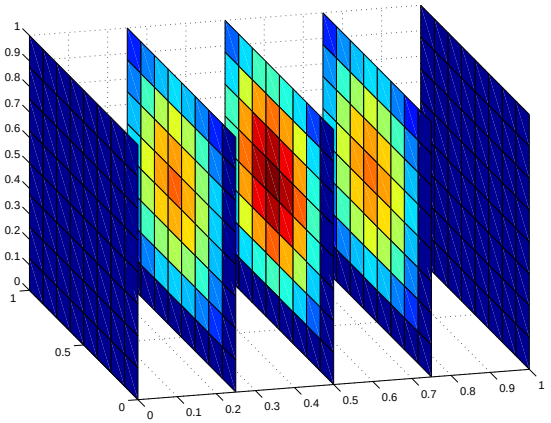
We were successful in correctly implementing the refinement procedure of the tetrahedralization. Also, the finite element method gave correct results in approximating the solution of (30). Unfortunately, despite attempts to solve the time-dependent system using Euler forward, we were unable to obtain correct results.

## 8.1 Suggestions for future work

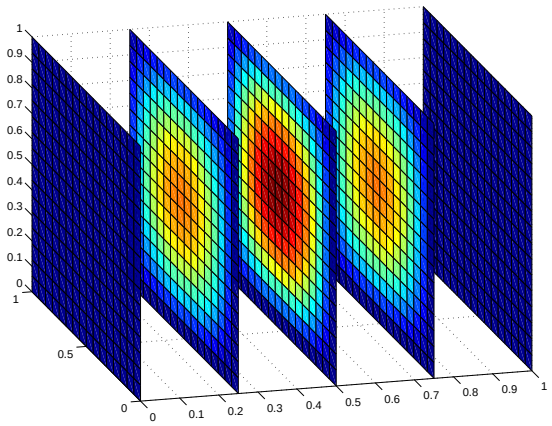
- Correctly implement the Euler forward method to solve the time-dependent boundary value problem.
- Try to visualize the solution of the system of ODEs by using Matlab's `movie` and `slice`.



(a) Initial tetrahedralization

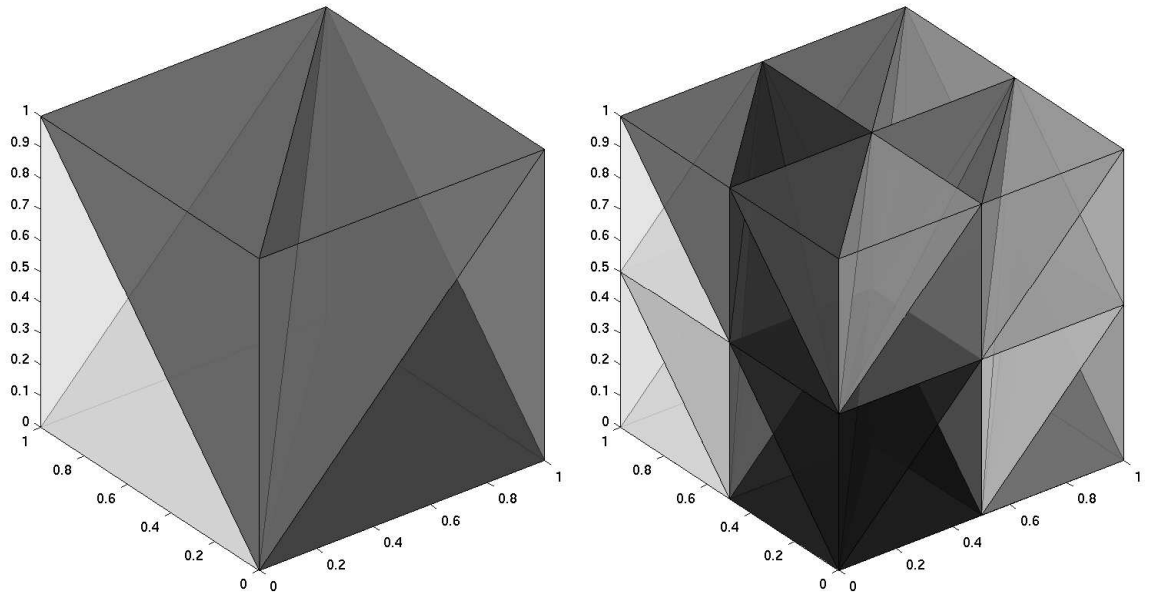


(b) After one refinement



(c) After two refinements

Figure 3: Slices of the approximation $\widehat{u}_h$ for two refinements



(a) Initial tetrahedralization

(b) After one refinement

Figure 4: Tetrahedralization of the unit cube

# A Appendix: the data structure

In this section we shall briefly describe the data structure we use for representing the tetrahedra. Tetrahedra are defined by four distinct vertices in 3-dimensional space which do not lie on the same (two-dimensional) plane. Therefore, it seems prudent to create a data structure which in one matrix keeps track of the coordinates of these vertices; while on the other hand we have a matrix keeping track about which vertices given tetrahedra are built from.
However, in our application this alone is not enough. We also need to keep seperate the boundary vertices, as the need special treatment regarding boundary conditions. In this appendix we will first introduce the tetrahedron storage mechanism. Then we will give a way to refine a given tetrahedra. After that, we will move to introduce a method for fast tetrahedron refining.

## A.1 Storage mechanism

Because we want the internal and external vertices strictly divided, we use instead of one matrix containing the vertex coordinates, two matrices. We denote the first coordinate matrix $N$, and in this matrix is stored the internal vertices. The coordinate vectors are stored column-wise; thus the matrix dimensions are 3 rows by $n_i$ columns, where $n_i$ the number of internal vertices. The second matrix is denoted $E$ and contains the external vertices (the vertices which lie on the boundary). The coordinate vectors are stored in the same way as in $N$. Using two matrices has as an advantage we immediatly know which vertices are boundary and which are not. Also, the matrices can grow independantly of each other.
Now the matrix which stores the individual tetrahedra. This matrix is denoted $T$ and stores

coordinate indeces column wise. Given that tetrahedra are defined by 4 points, $T$ has 4 rows and $n_t$ columns, $n_t$ the number of tetrahedra.
There is, however, a small problem. Suppose $t$ is a column from $T$; i.e., let $t$ define a single tetrahedron. Denote $t = \{t_1, t_2, t_3, t_4\}$. The $t_i$ should denote vertex indeces. However, we now use two coordinate matrices to store the vertices. To solve this problem, we define that for all $t_i > 0$, $t_i$ refers to an internal vertex. Thus $t_i$ is a column index of the matrix $N$. If, however, $t_i < 0$, we define that $t_i$ then points to an boundary vertex point in the matrix $E$. The column index is then given by $-t_i$.
Since we want to implement FEM with quadratures, we do not only need the vertices of the tetrahedron, but also the mid-points between all vertex pairs. We will, at the start of each FEM call, start an algorithm which adds all mid-points to $N$ or $E$, whichever is appropiate. The algorithm will return a matrix $M$, of size (6,$n_t$); at each column are index numbers to the mid-points beloning to the tetrahedron in $T$ at the same column index. The order at which the mid-points are stored row-wise is the same is in Figure 1
This concludes the basic structure of the tetrahedron storage mechanism.

### A.2 Tetrahedron refinement

Refining a tetrahedron can be done by dividing it into eight smaller tetrahedra, which are isomorphic to the original tetrahedron. To demonstrate this method, suppose we have an arbitrary tetrahedron defined by the vertices $t_1, \ldots, t_4$. Let us (arbitrarily) define the bottom triangle of this tetrahedron to be defined by the vertices $t_1$, $t_2$ and $t_3$. The top of the tetrahedron will then ofcourse be $t_4$.
Ofcourse, before we can refine, we must first construct new points on the edges of the current tetrahedron. We do this by creating a new point at the middle of each edge of the original tetrahedron. This makes it possible to construct isomorphic tetrahedra as a refinement. We group these new mid-point vertices in two; we have the lower plane mid-points which can be constructed by taking the mid-points of the edges at the lower triangle. The upper plane mid-points are constructed by taking the midpoints of the edges $t_1$-$t_4$, $t_2$-$t_4$, and $t_3$-$t_4$. Why we would group the new mid-points like this will become clear later on.
The first four refined tetrahedra are easy to construct. We take any of the four original vertices and let them define a new tetrahedron by connecting it with the three new mid-points found on its edges to the other original points. Now we have to fill up the tetrahedron with four other new small tetrahedra.
This can be done by choosing a vertex on the lower plane midpoints. Firstly, we then can create a tetrahedron with as base triangle the upper plane midpoints, and as the top the chosen vertex. If we now picture the not-yet completed refinement, we notice three open triangles on the original tetrahedron sides. If we take those triangles as a bottom and the chosen vertex as a top, we can create two new tetrahedron; for one triangle it is not possible to have the chosen vertex as the top, since that vertex is part of the base triangle. For that tetrahedron, we instead take the point on the upper plane midpoints which is symmetric to the chosen vertex as the top vertex. Now we have a total of eight tetrahedra and the refinement is complete.

### A.3 Refinement procedure

With the above information, we can now derive a refinement procedure. Let $t_0 = \{t_{0_1}, t_{0_2}, t_{0_3}, t_{0_4}\}$ again denote a tetrahedron. Let $l = \{l_1, l_2, l_3\}$ be the new lower plane midpoints, and $u = \{u_1, u_2, u_3\}$ be the new upper plane midpoints. Furthermore, let $m$ be the operator used to determine mid-points and let the following be true:

$$\begin{aligned} l_1 &= m(t_{0_1}, t_{0_2}) \\ l_2 &= m(t_{0_1}, t_{0_3}) \\ l_3 &= m(t_{0_2}, t_{0_3}) \\ u_1 &= m(t_{0_1}, t_{0_4}) \\ u_2 &= m(t_{0_2}, t_{0_4}) \\ u_3 &= m(t_{0_3}, t_{0_4}) \end{aligned}$$

Then the initial four new tetrahedra are defined as:

$$\begin{aligned} t_1 &= \{t_{0_1}, l_1, l_2, u_1\} \\ t_2 &= \{t_{0_2}, l_1, l_3, u_2\} \\ t_3 &= \{t_{0_3}, l_2, l_3, u_3\} \\ t_4 &= \{t_{0_4}, u_1, u_2, u_3\} \end{aligned}$$

Having already analyzed how to create the other four tetrahedra, we can immediatly write down those also, using the above coordinate names:

$$\begin{aligned} t_5 &= \{u_1, u_2, u_3, l_1\} \\ t_6 &= \{l_2, u_1, u_3, l_1\} \\ t_7 &= \{l_3, u_2, u_3, l_1\} \\ t_8 &= \{l_2, l_3, l_1, u_3\} \end{aligned}$$

We can easily write the above equations in any program. The key is how to efficiently add the new coordinates and tetrahedra to the data structure. We insist that a refinement procedure on a tetrahedron is called by giving the tetrahedron's index number. That way, we can easily replace the original tetrahedron in $T$ with one of the eight resulting ones. The other seven can be appended at the end of the matrix $T$. Updating $E$ and $N$ is not that easy. There are two main problems.
The first is that we do not want to store any coordinates twice; otherwise, the same coordinate

would be stored six time, at maximum. That is not all too horrible for a few refinements, but after only about 3 refinements the overhead already is quite excessive.
The second problem is that we need to determine whether or not any new points are on the boundary. This is not all that easy to accomplish. We want to have a method of refining which works on all initial tetrahedralizations, whatever may be their shape.
We would like to tackle both problems with methods that are not too slow either; this rules out simply checking if coordinates already exist to solve the first problem, this is just too slow. Instead, we use lookup matrices to find if coordinates already exists, and if they are already on the boundary. We use two different lookup matrices, one for the external vertices and one for the internal. This is so, because their functionality differs; the latter primarily tackles the first problem, whereas the matrix tied to the external points needs to tackle the second problem.

### A.3.1 Internal vertices lookup matrix and its usage

New vertices are added at each refinement call on a tetrahedron. Therefore, the lookup matrix must be given to and updated by the method that does the actual refining of one tetrahedron. Let us for now assume that the lookup matrix is globally available and that it is denoted by *look*. This matrix will have the initial size $(n_i + n_e, n_i + n_e)$, i.e., a square matrix of length equal to the sum of the internal and external vertices. All entries are initialized to zero. Now, when a new point between two vertices, the corresponding matrix entry is set to the index of that new point.
So when we want to check if a given coordinate already is added to the data structure, we check if $look(i_1, i_2)$ is zero (with $i_{1,2}$ the spawning points of the new mid-point). If so, this is a new coordinate and must be added to $N$. If not, we know that the coordinate is already added and furthermore, we even know the index at which this coordinate was added.
Note that *look* also has room for the external vertices. This is because not every midpoint derived from one or even two external points is an external point itself. This most of the time is not the case.

### A.3.2 External vertices lookup matrix and its usage

Denote the external lookup matrix $C$. $C$ is also a square matrix, with initial length equal to $n_e$, the number of external vertices. Now, $C$ differs from *look* in the sense that it is more of a connection matrix than a lookup matrix. It defines which external vertices lie on the same side of the boundary surface.
Now suppose we have calculated a new mid-point coordinate from parent coordinates from $E$ with indeces $i_{1,2}$ (If one of the coordinates does not come from $E$, the new midpoint surely is not a new point on the boundary). Now, if $C_{i_1} \cap C_{i_2} \neq \emptyset$, indicating that the parent coordinates are on the same side of the boundary surface, then the new coordinate surely is on the boundary as well. Thus we add this new coordinate to $E$ and return the new index $i_3$. However, we now also must update the $C$ matrix. For this, we set:

$$\forall x \in \{C_{i_1}\ capC_{i_2}\} : C(i_3, x) = 1$$

This procedure may be expressed in words as follows. The new point $i_3$ lies on the same side (or sides in case of corner coordinates) its parents have *in common*. In common, because

corner border coordinates may lie on multiple surfaces while others lie on exactly one side only. The in-between mid-point would then ofcourse lie on the second parent's side.

However, if we at a later point come across the same coordinate which must be added, we should not allow this process to repeat itself completely; this would cause multiple entries of the same coordinate. To prevent this, we use a second matrix *lookup* to administrate the already added points. This matrix behaves exactly like the *look* matrix for internal points discussed earlier, except this one obviously is used for external points.

Note also that the matrix $C$ contains data which never should be lost (that is, if we want to be able to refine further), and that the matrix $C$ may grow as more boundary coordinates will be added when refining. In contrast, the *look* and the *lookup* matrices are only necessary *during* the refinement procedure itself, and thus may be cleared after (or in between) refinement procedures.

### A.4 Conclusion

All the above ingredients put together make up our currently implemented data structure and refinement algorithm. We tried to find a method that allows faster determination when trying to find if a coordinate already has been added to the data structure. Also we have devised a way to correctly check if a new point derived from two boundary points is a boundary point itself.

## References

[1] Jan Brandts, Introduction to Numerical Analysis, Lecture Notes, February 6, 2006.

[2] Anders Logg, Automating the Finite Element Method, Lecture notes for the Sixth Winter School in Computational Mathematics, Toyota Technological Institute at Chicago, Geilo, March 5—10, 2006.