FINITE DIVIDED DIFFERENCES AND FINITE ELEMENTS FOR PARTIAL DIFFERENTIAL EQUATIONS: HOMEWORK 2

SKOUROLIAKOU VASILIKI

CONTENTS

LIST OF FIGURES

1 PROBLEM 1

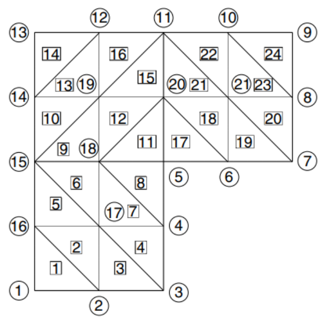
1.1 Finite Elements Method for Poisson Equation

Description: For the Poisson problem:

$$-\Delta u = 1, \quad (x,y) \in \Omega \subset \mathbf{R}^2, \quad u = 0 \ \text{ on } \ \theta\Omega,$$

where $\Omega = (-1,1)^2 \setminus (0,1) \times (-1,0)$, meaning a L-shape space. Find the solution of the above elliptic problem, considering the mesh in 1 and using the Finite Elements Method (FEM). Then try to resolve the problem, after thickening the mesh.

1.2 Solution



−1.0	−1.0
−0.5	−1.0
0	−1.0
0	−0.5
0	0
0.5	0
1.0	0
1.0	0.5
1.0	1.0
0.5	1.0
0	1.0
−0.5	1.0
−1.0	1.0
−1.0	0.5
−1.0	0
−1.0	−0.5
−0.5	−0.5
−0.5	0
−0.5	0.5
0	0.5
0.5	0.5

Coordinates

2	16	1
16	2	17
3	17	2
17	3	4
17	15	16
15	17	18
4	18	17
18	4	5
19	15	18
15	19	14
20	18	5
18	20	19
12	14	19
14	12	13
11	19	20
19	11	12
6	20	5
20	6	21
7	21	6
21	7	8
21	11	20
11	21	10
8	10	21
10	8	9

Elements

Figure 1: The mesh,nodes and connectivity matrices

1.2.1 *Describe the Finite Elements Method*

First we need to write the above problem in weak form. We introduce the space $\mathbf{V}$ to be the family of square-integrable functions $v : \Omega \to \mathbf{R}$, which have a weak derivative in Ω and satisfy the Dirichlet boundary condition v=0 on $\theta\Omega$.

We multiply the PDE by a test function $v \in \mathbf{V}$:

$$-\Delta u v = f v$$

and then we integrate over Ω:

$$-\int_\Omega \Delta u v dV = \int_\Omega f v dV$$

which can be written using the divergence Theorem:

$$\int_\Omega \nabla u \cdot \nabla v dV - \int_{\theta\Omega} v \nabla u \cdot \vec{n} dS = \int_\Omega f v dV$$

for all $v \in \mathbf{V}$, where $\vec{n}$ is the unit outward normal vector to the boundary $\theta\Omega$. However, v=0 on the boundary for all $v \in \mathbf{V}$, so the above relationship is written:

$$\int_\Omega \nabla u \cdot \nabla v dV = \int_\Omega f v dV.$$

So, the original problem can be transformed to the following in weak form:

Find $v \in \mathbf{V}$ such that $\int_\Omega \bigtriangledown u \cdot \bigtriangledown v dV = \int_\Omega fv dV$. for all $v \in \mathbf{V}$

At this point we should restrict the family of solutions to a smaller one $V_h \subset V$. To do so, we divide the domain Ω into triangles T, which from now on are the elements. The set of triangles, i.e. the mesh, is denoted by $\mathcal{T}$. The vertices of the triangles consist the nodes.

So, the family V_h consists of the functions that are continuous and linear on every element T. We also consider the basis functions $\phi's$ of V_h, which are "pyramid" functions being equal to one at each node i and zero everywhere else, such that any function $w_h \in V_h$ that is continuous in Ω and linear at each T can be written:

$$w_h = \sum_{i=1}^{N} W_i \phi_i$$

where W_i is the value of the function at node i. Hence, we solve the approximate problem:

Find $u_h \in V_h$ such that $\int_\Omega \bigtriangledown u_h \cdot \bigtriangledown v_h dV = \int_\Omega f v_h dV$. for all $v \in \mathbf{V}$, for all $v_h ob\mathbf{V}$.

Since $v_h \in V_h$ can be written as linear combination of "pyramid" functions. Similarly, $u_h \in V_h$ and it can also be written as linear combination of "pyramid" functions like:

$$u_h = \sum_{j=1}^{N} U_j \phi_j$$

, for some $U_j \in \mathbf{R}$. Eventually, the problem can be written:

Find U_j, $j = 1, .., N$, such that: $\sum_{j=1}^{N} U_j \int_\Omega \bigtriangledown \phi_j \cdot \bigtriangledown \phi_i dV = \int_\Omega f \phi_i dV$, for all i=1,...,N.

We observe that this is a linear system of N equations and N unknowns of the form: $AU = F$, where

$$a_{ij} = \int_\Omega \bigtriangledown \phi_j \cdot \bigtriangledown \phi_i dV, \quad F_i = \int_\Omega f \phi_i dV$$

and $U = (U_1, ..., U_n)^T$.

The above is the finite elements method.

1.2.2 *Implementation in C*

In this section we will present the C code used to implement FEM for the above problem. We will use the method in (9.32) in notes to compute the local stiffness matrix, i.e. the stiffness matrix concerning each one triangle in the mesh.

For each triangle we need to compute the quantity:

$$\begin{gathered}\int_T \bigtriangledown \phi_i \cdot \bigtriangledown \phi_j dx dy = \\ \int_{\hat{T}} [(B^{-T} \hat{\bigtriangledown} \hat{\phi_i}) \cdot (B^{-T} \hat{\bigtriangledown} \hat{\phi_j})] \times |det(B)| \, d\hat{x} d\hat{y} = \\ 0.5 \times |det(B)| \times [(B^{-T} \hat{\bigtriangledown} \hat{\phi_i}) \cdot (B^{-T} \hat{\bigtriangledown} \hat{\phi_j})] = \\ 0.5 \times |det(B)| \times [(B^T \setminus \hat{\bigtriangledown} \hat{\phi_i}) \cdot (B^T \setminus \hat{\bigtriangledown} \hat{\phi_j})]\end{gathered}$$

For the reference triangle:

$$\hat{\phi_1} = \hat{x}, \ \ \hat{\phi_2} = \hat{y}, \ \ \hat{\phi_3} = 1 - \hat{x} - \hat{y}$$

so

$$\hat{\bigtriangledown} \hat{\phi_1} = [1 \ 0]', \ \ \hat{\bigtriangledown} \hat{\phi_2} = [0 \ 1]', \ \ \hat{\bigtriangledown} \hat{\phi_3} = [-1 \ -1]'.$$

At this point we should also declare the transformation matrix B:

$$B = \begin{bmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{bmatrix}$$

so,

$$B^T = \begin{bmatrix} x_1 - x_3 & y_1 - y_3 \\ x_2 - x_3 & y_2 - y_3 \end{bmatrix}$$

Following, the routine to construct local stiffness matrix is presented. The routine uses the Gauss elimination procedure from https://www.gnu.org/software/gsl/.

```
void local_stiffness(
  const double array_b_T[][2],  /transpose transformation matrix B/
  const double array_b[][2],    /transformation matrix B/
  double det,                 /determinant of transformation B matrix/
  double array_local[][3],    /local stiffness matrix/
  double vector_local[3])     /local right-hand side vector b/
{
    int i,j,res;
    double area=0.5;
    double x1[2], x2[2], x3[2], phi[3], b_gsl[2];
    double inner_prod;
    double d_lambda[2][3]={ {1,0,-1} , {0,1,-1} };
    double lambda[2][3]={ {1,0,0} , {0,1,0} };

    for (i=0; i<3; i++){
    b_gsl[0]=d_lambda[0][i];
        b_gsl[1]=d_lambda[1][i];
        res=gauss_elimination_gsl_2(array_b_T, b_gsl, x1);
        if (res!=0){
            printf("there was an error in solving the system\n");
            return ;
        }
        for (j=0; j<3; j++){
            b_gsl[0]=d_lambda[0][j];
            b_gsl[1]=d_lambda[1][j];
            res=gauss_elimination_gsl_2(array_b_T, b_gsl,x2);
            if (res!=0){
                printf("there was an error in solving the system\n");
                return ;
            }
            inner_prod=x1[0]*x2[0]+x1[1]*x2[1];
            array_local[i][j]=area*det*inner_prod;
            }
        b_gsl[0]=lambda[0][j];
        b_gsl[1]=lambda[1][j];
        res= gauss_elimination_gsl_2(array_b, b_gsl, x3);
        if (res!=0){
            printf("there was an error in solving the system\n");
            return ;
        }
        phi_value(x3,phi);
        vector_local[i]=area*det*phi[i];
  }
}
```

Routine 1: local_stiffness routine

So, we run a double for loop to compute the above quantity for all combinations of i and j, i,j=1,2,3 and form the 3×3 local stiffness matrix. For each iteration of the double loop we compute two column vectors of length two (i.e. the quantities $(B^T \setminus \hat{\bigtriangledown}\hat{\phi}_i)$, $(B^T \setminus \hat{\bigtriangledown}\hat{\phi}_j)$), we find their inner product and we multiply it by $|\det(B)|$ and then by the constant area value(we explain later the integration procedure). Then this value is assigned to each position i,j of the local stiffness array.

At the loop over i's we compute the right-hand side b vector too, i.e. the quantity

$$\int_{\hat{T}} f \times B^{-1}\hat{\phi}_i \times |\det(B)| \, d\hat{x}d\hat{y}$$

where $\hat{x}, \hat{y}$ are the coordinates of each node multiplied by the transformation matrix. So, the value of the operation $area \times det \times phi_i$ is assigned to each position of the right side vector.

The integrals over the reference triangle region are computed as follows:

$$\int_D \int f \times (B^{-1}\hat{\phi}_i) \times |\det(B)| \, d\hat{x}d\hat{y} = \int_0^1 \int_0^{1-y} f \times (B \setminus \hat{\phi}_i) \times |\det(B)| \, d\hat{x}d\hat{y} =$$
$$f \times (B \setminus \hat{\phi}_i) \times |\det(B)| \times \int_0^1 \int_0^{1-y} d\hat{x}d\hat{y} = f \times (B \setminus \hat{\phi}_i) \times |\det(B)| \times \int_0^1 [x]_0^{1-y} d\hat{y} =$$
$$f \times (B \setminus \hat{\phi}_i) \times |\det(B)| \times [y - \frac{y}{2}]_0^1 = f \times (B \setminus \hat{\phi}_i) \times |\det(B)| \times 0.5.$$

In analogous way the integral on the left-hand side is computed.

Another fundamental function is the one that constructs the global stiffness matrix, say A. The parameter list of that function consists of two output parameters: the global stiffness matrix of size $N \times N$, here 21×21 and the global right side vector, denoted by b of size also N, i.e. 21.

```
void construct_global_matrix(double A_global[][N], double b[N]){
    int i,j,k,global_node[3];
    double x[3],y[3],det;
    double array_b[2][2],array_b_T[2][2], array_local[3][3], vector_local[3];

    for (k=0; k<NT; k++){
        find_cord_of_nodes(k,x,y,global_node);
        construct_b(x,y,array_b);
        construct_b_T(x,y,array_b_T);
        det=find_det(array_b);
        local_stiffness(array_b_T,array_b,det,array_local,vector_local);
        for (i=0; i<3; i++){
            b[global_node[i]-1]=b[global_node[i]-1]+vector_local[i];
            for (j=0; j<3; j++){
                A_global[global_node[i]-1][global_node[j]-1]= A_global[global_node[i]-1][
    global_node[j]-1] + array_local[i][j];
            }
        }
    }
}
```

Routine 2: construct_global_matrix

The function runs a main for loop over all elements, here the number of elements is the number of triangles, i.e. 24. Within this main for loop there is a second double for loop for i,j=1,2,3 that assigns the values of local stiffness matrix computed for each one triangle-element to the appropriate positions in the global matrix A. For example if a triangle consists of the nodes 2,16,1, i.e. the first one in Elements array, then it contributes to the following positions of global matrix: (2,2), (2,16), (2,1), (16,2), (16,16), (16,1), (1,2), (1,16), (1,1). So the inner double for loop will write the value of [i,j] element of local stiffness matrix to the corresponding positions of global array. At the same time the first inner for loop over i's constructs the right side vector correspondingly. Note that every time a node is met, the function adds the new value to it's previous one. Each node may be part of different triangles, so it finally consists of more than one contributions.

The indexes i,j of the global matrix A and global vector b, are taken from a $global\_node$ vector of length 3. This vector contains the nodes for each element-triangle, i.e. in the first iteration will contain 2,16,1. The double loop makes all the combinations of the $global\_node$ vector elements. Note that in C the index numbering starts from zero, that explains the -1 term at the global matrix indexing.

At this point the global stiffness matrix and the right side vector have been computed and we can solve the linear system $Au = b$ to find the solution u.

Before that, we need to impose the boundary conditions. From the problem definition we know that the value of the solution is equal to zero in the boundary (Dirichlet boundary condition), i.e. for the nodes 1,...,16. So, for this nodes we set the diagonal element of global stiffness matrix equal to one all the other elements of that row equal to zero. Also we set the corresponding index in right side vector b equal to zero.

```
/* mark nodes on the boundary,to impose Dirichlet boundary condition */
    for (i=0; i<N; i++){
        if ( (nodes[i][1]==-1) || (nodes[i][0]==-1) || (nodes[i][1]==1) || (nodes[i][0]==1) || (
    nodes[i][1]==0 && nodes[i][0]>=0) || (nodes[i][0]==0 && nodes[i][1]<0) ){
            cond_d[i]=DIRICHLET;
        }
        else {
            cond_d[i]=inner_node;
        }
    }
    construct_global_matrix(A_global,b);
/*impose boundary condition*/
    for (i=0; i<N; i++){
        if (cond_d[i]==DIRICHLET) {
            b[i]=0;
        for (j=0; j<N; j++){
            A_global[i][j]=0;
        }
        A_global[i][i]=1;
    }
}
```

Routine 3: part of main function

Now we are ready to solve the system.
The solution u is shown in the following figure:

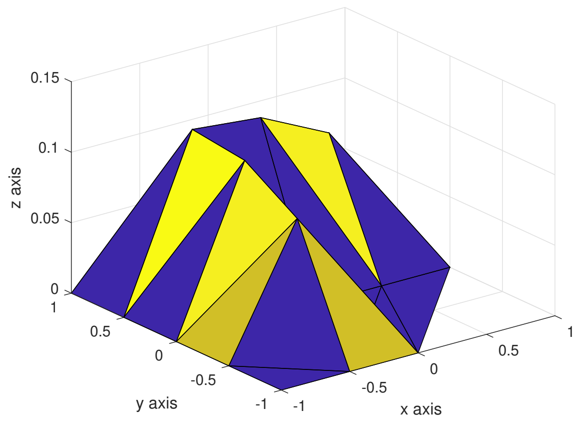


Figure 2: The solution u obtained by FEM

1.2.3 *Mesh Thickening*

At this section we aim to implement the above FE method over a thicker mesh, i.e. the same L space consisting of more elements and nodes. The triangles now will be obviously smaller. Specifically, we make four triangles out of each triangle of the original mesh. For this purpose, we find the medium of each side of each original triangle and then connect the three mediums. Each medium consists a new node in the new mesh and now each original triangle is divided in four new triangles. The coordinates of each new nodes are the coordinates of the medium, and the nodes array is updated correspondingly. The new mesh is obtained from a C function that takes as input parameters the original nodes and connectivity arrays and produces the new ones implementing the procedure described above. The following pictures shows the bottom of the solution surface for the new mesh. The second figure shows the solution u.

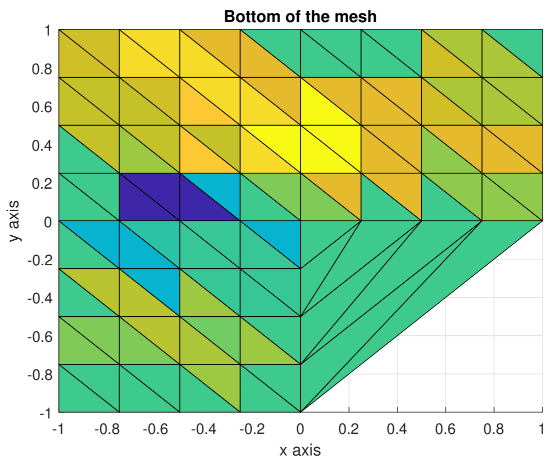


Figure 3: Bottom of the solution over the thicker mesh

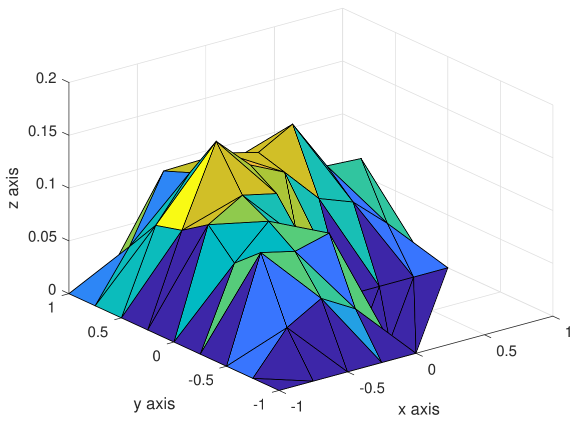


Figure 4: Solution over the thicker mesh

2 PROBLEM 2

2.1 Heat Equation

For the parabolic problem:

$$u_t - \Delta u = 1$$

, (t,x,y)$\in [0,1] \times \Omega \subset$ R² and u=0 at $(0,1) \times \theta\Omega$, where Ω is the same as the previous problem, find the solution u by defining:

2.1.1 *An Implicit Euler Method*

2.1.2 *An Crank-Nicolson Method*

2.2 Solution

First we aim to write the above problem in weak form [1], so we multiply both sides with a test function $v \in \mathbf{V}$, where $\mathbf{V} \equiv H^1_0((\Omega))$, i.e. the family of function that have a first weak derivative in Ω and are equal to zero on $\theta\Omega$. We also integrate over the domain Ω at the same time:

$$\int_\Omega u_t v dxdy - \int_\Omega \Delta u v dxdy = \int_\Omega f v dxdy = \longrightarrow$$
$$\int_\Omega u_t v dxdy + \int_\Omega \bigtriangledown u \cdot \bigtriangledown v dxdy - \int_{\theta\Omega} v \bigtriangledown u \cdot \vec{n} dS = \int_\Omega f v dxdy = \longrightarrow$$
$$\int_\Omega u_t v dxdy + \int_\Omega \bigtriangledown u \cdot \bigtriangledown v dxdy = \int_\Omega f v dxdy$$

where we made again use of the divergence theorem and the fact that v=0 on $\theta\Omega$. So the above is the weak form of the problem for all $v \in \mathbf{V}$.

2.2.1 *Implicit Euler Method*

In order to define an implicit Euler method, we first need a uniform subdivision of time space $[0, T_f = \equiv [0,1]$, with a time step $\tau = T_f/N_t$ so: $0 = t_0 < t_1 = t_0 + \tau < ... < t_n = T_f$

We use backward divided difference to approximate u_t:

$$u_t(x,t) = \approx \frac{u(t,x) - u(t-\tau, x)}{\tau} = \frac{u^n - u^{n-1}}{\tau}.$$

Hence, if we replace that to the above weak form of the problem we get:

$$\int_\Omega \frac{u^n - u^{n-1}}{\tau} dx + \int_\Omega \bigtriangledown u^n \cdot \bigtriangledown v dxdy = \int_\Omega f v dxdy$$

for all $v \in \mathbf{V}$.

Now, we restrict the family of functions $v \in \mathbf{V}$ to a smaller one, let $\mathbf{V_h} \subset \mathbf{V}$ such that every $v_h \in \mathbf{V_h}$ is a pyramid function ϕ_i. Hence, we write the above problem again:

Find $u_h \in \mathbf{V_h}$ such that: $\int_\Omega \frac{u_h^n - u_h^{n-1}}{\tau} v_h dxdy + \int_\Omega \bigtriangledown u_h^n \cdot \bigtriangledown v_h dxdy = \int_\Omega f v_h dxdy$,

or after multiplying by τ and rearrangement:

$$\int_\Omega u_h^n v_h dxdy + \tau \int_\Omega \bigtriangledown u_h^n \cdot \bigtriangledown v_h dxdy = \tau \int_\Omega f v_h dxdy + \int_\Omega u_h^{n-1} v_h dxdy.$$

And taking into consideration that $\phi_i's$ consist a base of space $\mathbf{V_h}$ we can write u_h like:

$$u_h = \sum_{j=1}^{N_x} u_{h_j}\phi_j$$

and then we can write the problem at the form:

$$\sum_{j=1}^{N_x} u_j^n \int_\Omega \phi_j\phi_i dxdy + \tau \sum_{j=1}^{N_x} u_j^n \int_\Omega \nabla\phi_j \cdot \nabla\phi_i dxdy = \tau \int_\Omega f\phi_i dxdy + \sum_{j=1}^{N_x} u_j^n \int_\Omega \phi_j\phi_i dxdy.$$

for $i = 1, ..., N_x$ and $j = 1, .., N_t$.

We define the mass matrix as follows:

$$m_{ij} = \sum_{j=1}^{N_x} \int_\Omega \phi_j\phi_i dxdy$$

,so in matrix the problem is written:

$$(M + \tau A)U^n = \tau F + MU^{n-1}. \quad (1)$$

i.e. a linear system with respect to U^n, with the right-hand side depending on the previous time value.

2.2.2 *Implementation in C*

We are going to use the code we presented in previous problem and add what is needed in order to find the solution vector U^n, as described above. The changes will be presented in this section:

First we need to compute mass matrix M. We follow the exact same process as with stiffness matrix A in `local_stiffness` routine presented above. The difference is that for mass matrix we need to compute the following quantity for every triangle in the mesh:

$$\int_T \phi_j \cdot \phi_i dxdy =$$
$$\int_{\hat{T}} [(B^{-1}\phi_j) \cdot (B^{-1}\phi_i)\ ] \times |det(B)|\ d\hat{x}d\hat{y}$$
$$0.5 \times |det(B)| \times [(B \setminus \phi_j) \cdot (B \setminus \phi_i)].$$

, where $phi_1 = \hat{x}$, $\phi_2 = \hat{y}$, $\phi_3 = 1 - \hat{x} - \hat{y}$.

Hence, in `local_stiffness` routine we also construct a local mass matrix of size 3×3 and we fill it in the same way as local stiffness matrix A.

We also need the τ value to construct the system of the problem, which is computed by the formula: $\tau = T_f/N_t$, where $T_f = 1$ and we chose to solve the problem for $N_t = 100$, hence $\tau = 0.01$.

We also observe that now we need to solve a linear system with respect to U^n for every n (see (1), with $u_i^0 = 0$. The left-hand side of the system is equal to $(M + \tau A)$ and the right-hand side is equal to $\tau F + MU^{n-1}$, where U^{n-1} is computed from the previous step.

So, we need a for loop in main function over n=1,..,100. For each iteration we solve a linear system using Gauss elimination method as in the previous problem. The boundary conditions are imposed according to problem 1.

```c
/* mark nodes on the boundary,to impose Dirichlet boundary condition */
    for (i=0; i<N; i++){
        if ( (nodes[i][1]==-1) || (nodes[i][0]==-1) || (nodes[i][1]==1) || (nodes[i][0]==1) || (
    nodes[i][1]==0 && nodes[i][0]>=0) || (nodes[i][0]==0 && nodes[i][1]<0) ){
            cond_d[i]=DIRICHLET;
        }
        else {
            cond_d[i]=inner_node;
        }
    }
    construct_global_matrix(A_global,b,M_global);
    construct_system_left(A_global, M_global, taf, A_2_global);
    construct_system_right(b,M_global,taf,u,b_2);
/*impose boundary condition*/
    for (i=0; i<N; i++){
        if (cond_d[i]==DIRICHLET) {
            b_2[i]=0;
            for (j=0; j<N; j++){
                A_2_global[i][j]=0;
            }
            A_2_global[i][i]=1;
        }
    }
/*loop for all n*/
    for (i=1;i<Nt;i++){
        res=gauss_elimination_gsl(A_2_global,b_2,u);
        if (res!=0){
            printf("there was an error in solving the system\n");
            return -1;
        }
        construct_system_right(b, M_global,taf,u,b_2);
        for (j=0; j<N; j++){
            if (cond_d[j]==DIRICHLET) {
                b_2[j]=0;
            }
        }
        for (j=0;j<N; j++){
            u_heat[i][j]=u[j];
        }
    }
```

Routine 4: loop in main function

We save the solution for every n in u_heat matrix.
We chose to plot the solution for n=20,50,100:

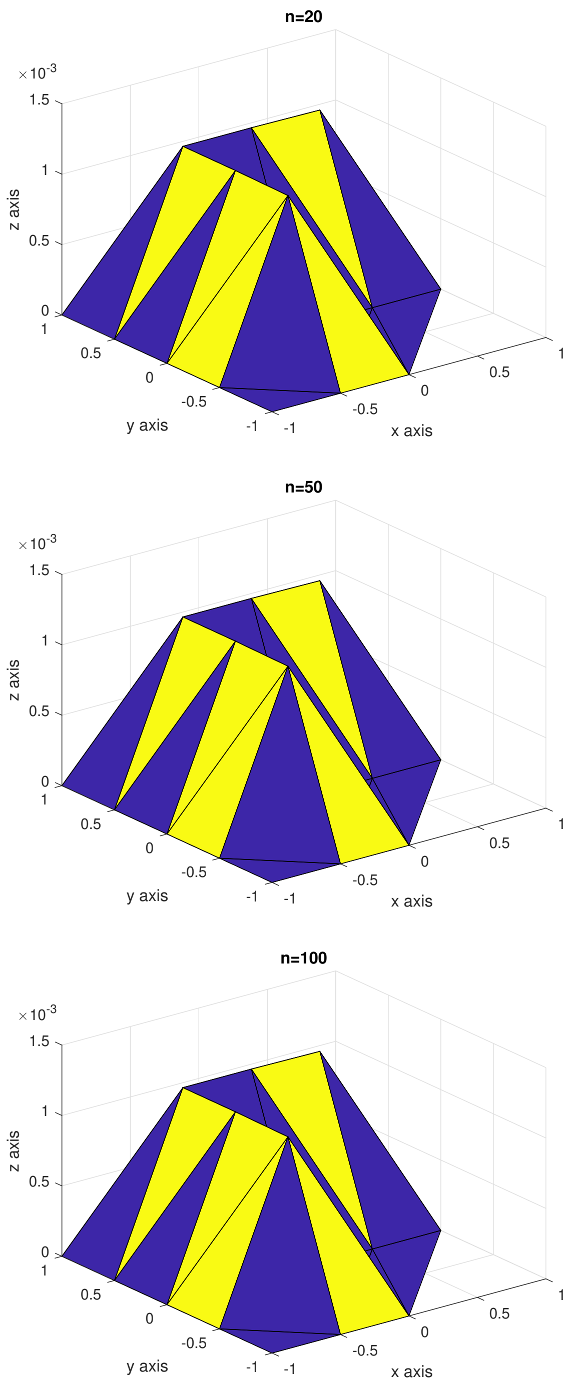


Figure 5: The solution u using Implicit Euler method for n=20,50,100

2.2.3 *A Crank-Nicolson method*

In order to define a Crank-Nicolson method for the heat equation, we first need to admit that whatever we discussed for the Implicit Euler Method above is true for Crank-Nicolson method too. The difference is that we will write the weak form as follows: Find $v_h \in \mathbf{V_h}$ such that:

$$\int_\Omega \frac{u^n - u^{n-1}}{\tau} dx + \frac{1}{2}\int_\Omega \nabla u^n \cdot \nabla v dxdy + \frac{1}{2}\int_\Omega \nabla u^{n-1} \cdot \nabla v dxdy = \int_\Omega f v dxdy$$

for all $v_h \in \mathbf{V_h}$. After multiplying by τ and rearrange we end up to the following problem:

$$\int_\Omega u_h^n v_h dxdy + \frac{\tau}{2}\int_\Omega \nabla u_h^n \cdot \nabla v_h dxdy = \tau\int_\Omega f v_h dxdy + \int_\Omega u_h^{n-1} v_h dxdy - \frac{\tau}{2}\int_\Omega \nabla u_h^{n-1} \cdot \nabla v_h dxdy.$$

The above problem can be written, according to the Implicit Euler method above, in matrix form:

$$(M + \frac{\tau}{2}A)U^n = \tau F + MU^{n-1} - \frac{\tau}{2}AU^{n-1}. \qquad (2)$$

i.e. again a linear system with respect to U^n, with the right-hand side depending on the previous time value.

This method allows us to take bigger time steps for the solution to converge, so we are are going to set τ equal to 0.1 and use $N_t = 10$. We chose to plot the solution for n=1,5,10.

2.2.4 *Implementation in C*

The implementation is exactly the same, the only thing changed is the way the left-hand and right-hand sides of the final system are constructed. (See Appendix for those routines).

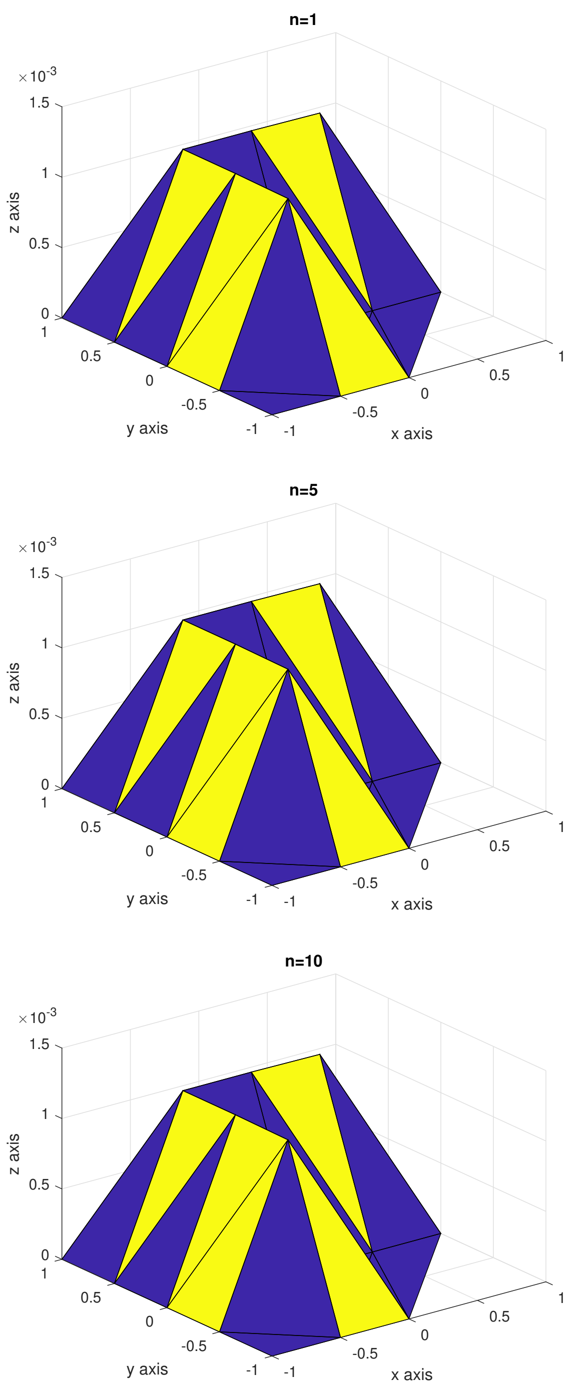


Figure 6: The solution u using Crank-Nicolson method for n=1,5,10

We observe that the solution converges even for a small number of steps.

3 APPENDIX

At this section we present the secondary routines called by the main ones presented above. A routine called by local_stiffness is phi_value, which takes as input parameter the column vector of transformed coordination of a node and returns a three position array with phi_i's values for i=1,2,3.

```c
void phi_value(const double x[2], double phi[3]){
    phi[0]= x[0];
    phi[1]= x[1];
    phi[2]= 1−x[0]−x[1];
}
```

Routine 5: phi_value

The following routines are called by construct_global_matrix main routine: First for every element the function find_cord_of_nodes is called. This takes as input parameter the number of element and returns as output parameters two vectors of length three with x and y coordinates for each vertex of triangle, taken from the connectivity array, and also returns the global_node vector we discussed before.

```c
void find_cord_of_nodes(const int elem, double x[3], double y[3], int global_node[3]){
int j,sel;

    for (j=0; j<3; j++){
        sel=connectivity[elem][j];
        global_node[j]=connectivity[elem][j];
        x[j]=nodes[sel −1][0];
        y[j]=nodes[sel −1][1];
    }
}
```

Routine 6: find_cords_of_nodes

After the coordinates of the vertices of each triangle are determined the routines to construct transformation matrix B and B^T are called.

The routines take as input parameters two vectors of length 3 , i.e. the coordinates of each vertex of each triangle and the output parameter is the transformation matrix B or B^T of size 2×2.

```c
void construct_b(const double x[3],const double y[3], double array_b_T[][2]){
  array_b[0][0]=x[0]−x[2];
  array_b[0][1]=y[0]−y[2];
  array_b[1][0]=x[1]−x[2];
  array_b[1][1]=y[1]−y[2];
}
```

Routine 7: construct_b_T

```c
void construct_b(const double x[3],const double y[3], double array_b[][2]){
  array_b[0][0]=x[0]−x[2];
  array_b[0][1]=x[1]−x[2];
  array_b[1][0]=y[0]−y[2];
  array_b[1][1]=y[1]−y[2];
}
```

Routine 8: construct_b

Then a routine to compute the determinant of B matrix is called. The input parameter is the matrix B and returns the determinant. Note that this function is also constructed only for 2×2 matrices.

```c
double find_det(const double array_b[][2]){
  return (fabs(array_b[0][0]∗array_b[1][1]−array_b[1][0]∗array_b[0][1]));
}
```

Routine 9: find_det

Next we present the routine called from main to solve the final system: The routine takes as input parameters the global stiffness matrix and right side vector and the output parameter is the solution, which is also printed.

```c
int elimination_2x2_gsl(const double A[][N], const double b[N], double u[N]){
    int s,i;

     gsl_matrix *A_m = construct_gsl_matrix(A);
     gsl_vector *b_v = construct_gsl_vector(b);
     gsl_vector *u_v = gsl_vector_alloc(N);
     gsl_permutation *p = gsl_permutation_alloc(N);

    if (A_m == NULL || b_v == NULL || u_v == NULL || p == NULL) {
        printf("Cannot construct matrix or/and vectors\n");
        return -1;
    }

    /* Solve the system*/
    gsl_linalg_LU_decomp(A_m, p, &s);
    gsl_linalg_LU_solve(A_m, p, b_v, u_v);

    /*Print the solution*/
    printf("\nu = \n");
    gsl_vector_fprintf (stdout, u_v, "%g");

    for (i = 0; i < N; i++) {
        u[i]=gsl_vector_get(u_v,i);
    }
    return 0;
}
```

Routine 10: *Gauss_elimination_gsl*

The library has its own type for matrices and vectors so we need the following function fr conversion:

```c
gsl_matrix *construct_gsl_matrix(const double A[][N]){
    int i,j;

    gsl_matrix *gsl_m = gsl_matrix_alloc(N,N);
    if (gsl_m == NULL) {
        return NULL;
    }

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            gsl_matrix_set(gsl_m, i, j, A[i][j]);
        }
    }

    return gsl_m;
}
```

Routine 11: Construct gsl matrix

```c
gsl_vector *construct_gsl_vector(const double b[N]){
    int i;

    gsl_vector *gsl_v = gsl_vector_alloc(N);
    if (gsl_v == NULL) {
        return NULL;
    }

    for (i = 0; i < N; i++) {
        gsl_vector_set(gsl_v, i, b[i]);
    }

    return gsl_v;
}
```

Routine 12: Construct gsl vector

Finally we present the Matlab script used to plot the solution u:

```
clear;clc;
load output.dat;
x = output(:,1) ; y =output(:,2) ; z = output(:,3);
dt = delaunayTriangulation(x,y) ;
tri = dt.ConnectivityList ;
figure
trisurf(tri,x,y,z);
xlabel("x axis");
ylabel("y axis");
zlabel("z axis");
```

Routine 13: Matlab script for plotting

In the second problem we use two extra routines to construct the left-hand and right-hand sides of the linear system for every step. The *construct_system_left* routine is responsible to construct the left-hand side, so it takes as input parameters the mass M and stiffness matrix A and the τ value and returns as output parameter a new *A_2_global*:

```
void construct_system_left(const double A[][N], const double M[][N], const double taf, double A_2
    [][N]){
    int res,i,j;

    gsl_matrix *A_m = construct_gsl_matrix(A);
    gsl_matrix *M_m = construct_gsl_matrix(M);
    if(A_m == NULL || M_m == NULL){
         printf("Cannot construct matrix or/and vectors\n");
        return;
    }

    res=gsl_matrix_scale(A_m,taf);
    if (res!=0){
        printf("Cannot multiply the vector with the constant factor\n");
        return;
    }
    res=gsl_matrix_add(A_m,M_m);
    if (res!=0){
        printf("Cannot multiply the vector with the constant factor\n");
        return;
    }
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A_2[i][j]=gsl_matrix_get(A_m,i,j);
        }
    }

}
```

Routine 14: construct_system_left

The second routine, *construct_system_right* takes as input parameters the mass M matrix, the τ value and the value of the solution in previous step n-1 and returns as output parameter the new vector b_2:

```
void construct_system_right(const double b[N], const double M[][N], const double taf, const
    double u[N], double b_2[N]){
    int i,j;
    double mu[N];
    int res,prod;

    prod=0;
    for (i=0; i<N; i++){
        for (j=0; j<N; j++){
            prod=prod+=M[i][j]*u[j];
        }
        mu[i]=prod;
        prod=0;
    }

    gsl_vector *b_v = construct_gsl_vector(b);
    gsl_vector *mu_v = construct_gsl_vector(mu);

    if(b_v == NULL || mu_v == NULL){
         printf("Cannot construct matrix or/and vectors\n");
        return;
    }
    res=gsl_vector_scale(b_v,taf);
    if (res!=0){
        printf("Cannot multiply the vector with the constant factor\n");
        return;
    }
    res=gsl_vector_add(b_v,mu_v);
    if (res!=0){
        printf("Cannot multiply the vector with the constant factor\n");
        return;
    }
    for (i = 0; i < N; i++) {
        b_2[i]=gsl_vector_get(b_v,i);
    }
}
```

Routine 15: construct_system_right

If we want to implement the Crank-Nicolson method the only change we need to make in construct_system_left is to multiply A_global matrix with $\tau/2$. In construct_system_right we just add the term: $\tau/2 \times A \times U^{n-1}$, i.e. we multiply the input column vector u with matrix A too, the scaling in by $-\tau/2$ and then adding the term to the previous value of b_2 vector.

REFERENCES

[1] Manolis Georgoulis. The finite element method for elliptic problems. In *Computational Methods for Partial Differential Equations, 2019*, pages 98–119, 2019.