A Matlab Tutorial for Diffusion-Convection-Reaction Equations using DGFEM

Murat Uzunca¹, Bülent Karasözen²

Abstract. We present a collection of MATLAB routines using discontinuous Galerkin finite elements method (DGFEM) for solving steady-state diffusion-convection-reaction equations. The code employs the sparse matrix facilities of MATLAB with "vectorization" and uses multiple matrix multiplications "MULTIPROD" [5] to increase the efficiency of the program.

Keywords. Discontinuous Galerkin FEMs, Diffusion-convection-reaction equations, Matlab

Preprint No. 2014-4
August 2014

¹Department of Mathematics, Middle East Technical University, 06800 Ankara, Turkey, email: uzunca@metu.edu.tr
²Department of Mathematics and Institute of Applied Mathematics, Middle East Technical University, 06800 Ankara, Turkey, email: bulent@metu.edu.tr
1 DG discretization of the linear model problem

Many engineering problems such as chemical reaction processes, heat conduction, nuclear reactors, population dynamics etc. are governed by convection-diffusion-reaction partial differential equations (PDEs). The general model problem used in the code is

\[
\begin{align*}
\alpha u - \varepsilon \nabla u &= f \quad \text{in } \Omega, \\
u &= g^D \quad \text{on } \Gamma^D, \\
\varepsilon \nabla u \cdot n &= g^N \quad \text{on } \Gamma^N. 
\end{align*}
\]

The domain \(\Omega\) is bounded, open, convex in \(\mathbb{R}^2\) with boundary \(\partial \Omega = \Gamma^D \cup \Gamma^N\), \(\Gamma^D \cap \Gamma^N = \emptyset\), \(0 < \varepsilon \ll 1\) is the diffusivity constant, \(f \in L^2(\Omega)\) is the source function, \(b \in (W^{1,\infty}(\Omega))^2\) is the velocity field, \(g^D \in H^{3/2}(\Gamma^D)\) is the Dirichlet boundary condition, \(g^N \in H^{1/2}(\Gamma^N)\) is the Neumann boundary condition and \(n\) denote the unit outward normal vector to the boundary.

The weak formulation of (1) reads as: find \(u \in U\) such that

\[
\int_{\Omega} (\varepsilon \nabla u \cdot \nabla v + b \cdot \nabla uv + \alpha uv) \, dx = \int_{\Omega} f v \, dx + \int_{\Gamma^N} g^N v \, ds, \quad \forall v \in V
\]

where the solution space \(U\) and the test function space \(V\) are given by

\[
U = \{u \in H^1(\Omega) : u = g^D \text{ on } \Gamma^D\}, \quad V = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma^D\}.
\]

Being the next step, an approximation to the problem (2) is found in a finite-dimensional space \(V_h\). In case of classical (continuous) FEMs, the space \(V_h\) is set to be the set of piecewise continuous polynomials vanishing on the boundary \(\partial \Omega\).

In contrast to the continuous FEMs, the DGFEMs uses the set of piecewise polynomials that are fully discontinuous at the interfaces. In this way, the DGFEMs approximation allows to capture the sharp gradients or singularities that affect the numerical solution locally. Moreover, the functions in \(V_h\) do not need to vanish at the boundary since the boundary conditions in DGFEMs are imposed weakly.

In our code, the discretization of the problem (1) is based on the discontinuous Galerkin methods for the diffusion part [1, 6] and the upwinding for the convection part [2, 4]. Let \(\{\xi_h\}\) be a family of shape regular meshes with the elements (triangles) \(K_i \in \xi_h\) satisfying \(\Omega = \bigcup K\) and \(K_i \cap K_j = \emptyset\) for \(K_i, K_j \in \xi_h\). Let us denote by \(\Gamma_0, \Gamma_D\) and \(\Gamma_N\) the set of interior, Dirichlet boundary and Neumann boundary edges, respectively, so that \(\Gamma_0 \cup \Gamma_D \cup \Gamma_N\) forms the skeleton of the mesh. For any \(K \in \xi_h\), let \(P_k(K)\) be the set of all polynomials of degree at most \(k\) on \(K\). Then, set the finite dimensional solution and test function space by

\[
V_h = \{v \in L^2(\Omega) : v|_K \in P_k(K), \forall K \in \xi_h\} \not\subset V.
\]
Figure 1: Two adjacent elements sharing an edge (left); an element near to domain boundary (right)

Note that the trial and test function spaces are the same because the boundary conditions in discontinuous Galerkin methods are imposed in a weak manner. Since the functions in $V_h$ may have discontinuities along the inter-element boundaries, along an interior edge, there would be two different traces from the adjacent elements sharing that edge. In the light of this fact, let us first introduce some notations before giving the DG formulation. Let $K_i, K_j \in \mathcal{K}_h$ ($i < j$) be two adjacent elements sharing an interior edge $e = K_i \cap K_j \subset \Gamma_0$ (see Fig.1). Denote the trace of a scalar function $v$ from inside $K_i$ by $v_i$ and from inside $K_j$ by $v_j$. Then, set the jump and average values of $v$ on the edge $e$

$$[v] = v_i n_e - v_j n_e, \quad \{v\} = \frac{1}{2} (v_i + v_j),$$

where $n_e$ is the unit normal to the edge $e$ oriented from $K_i$ to $K_j$. Similarly, we set the jump and average values of a vector valued function $\mathbf{q}$ on $e$

$$[\mathbf{q}] = \mathbf{q}_i \cdot n_e - \mathbf{q}_j \cdot n_e, \quad \{\mathbf{q}\} = \frac{1}{2} (\mathbf{q}_i + \mathbf{q}_j),$$

Observe that $[v]$ is a vector for a scalar function $v$, while, $[\mathbf{q}]$ is scalar for a vector valued function $\mathbf{q}$. On the other hands, along any boundary edge $e = K_i \cap \partial \Omega$, we set

$$[v] = v_i n, \quad \{v\} = v_i, \quad [\mathbf{q}] = \mathbf{q}_i \cdot n, \quad \{\mathbf{q}\} = \mathbf{q}_i,$$

where $n$ is the unit outward normal to the boundary at $e$.

We also introduce the inflow parts of the domain boundary and the boundary of a mesh element $K$, respectively

$$\Gamma^- = \{x \in \partial \Omega : \mathbf{b}(x) \cdot n(x) < 0\}, \quad \partial K^- = \{x \in \partial K : \mathbf{b}(x) \cdot n_K(x) < 0\}.$$

Then, the DG discretized system to the problem (1) combining with the upwind discretization for the convection part reads as: find $u_h \in V_h$ such that

$$a_h(u_h, v_h) = l_h(v_h) \quad \forall v_h \in V_h,$$

(3)
\[ a_h(u_h, v_h) = \sum_{K \in \mathcal{K}_h} \int_K \epsilon \nabla u_h \cdot \nabla v_h dx + \sum_{K \in \mathcal{K}_h} \int_K (b \cdot \nabla u_h + \alpha u_h) v_h dx \]

\[ - \sum_{e \in \Gamma_D \cup \Gamma_0} \int_e \{\epsilon \nabla u_h\} \cdot [v_h] ds + \kappa \sum_{e \in \Gamma_D \cup \Gamma_0} \int_e \{\epsilon \nabla v_h\} \cdot [u_h] ds \]

\[ + \sum_{K \in \mathcal{K}_h} \int_{\partial K \setminus \partial \Omega} b \cdot \mathbf{n} (u_h^{\text{out}} - u_h^{\text{in}}) v_h ds - \sum_{K \in \mathcal{K}_h} \int_{\partial K \setminus \partial \Gamma^-} b \cdot \mathbf{n} u_h^{\text{in}} v_h ds \]

\[ + \sum_{e \in \Gamma_D \cup \Gamma_0} \frac{\sigma}{h_e} \int_e [u_h] : [v_h] ds, \]

\[ l_h(v_h) = \sum_{K \in \mathcal{K}_h} \int_K f v_h dx + \sum_{e \in \Gamma_D} \int_e g_D \left( \frac{\sigma}{h_e} v_h - \epsilon \nabla v_h \cdot \mathbf{n} \right) ds \]

\[ - \sum_{K \in \mathcal{K}_h} \int_{\partial K \setminus \partial \Gamma^-} b \cdot \mathbf{n} g^D v_h ds + \sum_{e \in \Gamma_N} \int_e g^N v_h ds, \]

where \( u_h^{\text{out}} \) and \( u_h^{\text{in}} \) denotes the values on an edge from outside and inside of an element \( K \), respectively. The parameter \( \kappa \) determines the type of DG method, which takes the values \( \{-1, 1, 0\} \): \( \kappa = -1 \) gives "symmetric interior penalty Galerkin" (SIPG) method, \( \kappa = 1 \) gives "non-symmetric interior penalty Galerkin" (NIPG) method and \( \kappa = 0 \) gives "inconsistent interior penalty Galerkin" (IIPG) method. The parameter \( \sigma \in \mathbb{R}_{\geq 0}^{+} \) is called the penalty parameter which should be sufficiently large; independent of the mesh size \( h \) and the diffusion coefficient \( \epsilon \) [6] [Sec. 2.7.1]. In our code, we choose the penalty parameter \( \sigma \) on interior edges depending on the polynomial degree \( k \) as \( \sigma = 3k(k+1) \) for the SIPG and IIPG methods, whereas, we take \( \sigma = 1 \) for the NIPG method. On boundary edges, we take the penalty parameter as twice of the penalty parameter on interior edges.

## 2 Descriptions of the MATLAB code

The given codes are mostly self-explanatory with comments to explain what each section of the code does. In this section, we give the description of our main code. The use of the main code consists of three parts

1. Mesh generation,
2. Entry of user defined quantities (boundary conditions, order of basis etc.),
3. Forming and solving the linear systems,
4. Plotting the solutions.

Except the last one, all the parts above take place in the m-file `Main_Linear.m` which is the main code to be used by the users for linear problems without need to entry to any other m-file. The last part, plotting the solutions, takes place in the m-file `dg_error.m`. 

4
2.1 Mesh generation

In this section, we define the data structure of a triangular mesh on a polygonal
domain in $\mathbb{R}^2$. The data structure presented here is based on simple arrays [3]
which are stored in a MATLAB "struct" that collects two or more data fields in
one object that can then be passed to routines. To obtain an initial mesh, firstly,
we define the nodes, elements, Dirichlet and Neumann conditions in the m-file
Main_Linearm, and we call the getmesh function to form the initial mesh structure
mesh.

% Generate the mesh
% Nodes
Nodes = [0,0;0.5,0;1,0;0,0.5;0.5,1;0.5;0,1;0.5;1;1,1];
% Elements
Elements = [4,1,5;1,2,5;2,6;2,3,6;4,8;4,5,8;8,5,9;5,6,9];
% Dirichlet bdry edges
Dirichlet = [1,2;2,3;1,4;3,6;4,7;6,9;7,8;8,9];
% Neumann bdry edges
Neumann = [];
% Initial mesh struct
mesh = getmesh(Nodes,Elements,Dirichlet,Neumann);

As it can be understood, each row in the Nodes array corresponds to a mesh node
with the first column keeps the x-coordinate of the node and the second is for the
y-coordinate, and the $i$-th row of the Nodes array is called the node having index
$i$. In the Elements array, each row with 3 columns corresponds to a triangular ele-
ment in the mesh containing the indices of the nodes forming the 3 vertices of the
triangles in the counter-clockwise orientation. Finally, in the Dirichlet and Neu-
mann arrays, each row with 2 columns corresponds to a Dirichlet and Neumann
boundary edge containing the indices of the starting and ending nodes, respectively
(see Fig[2]). The mesh "struct" in the code has the following fields:

- Nodes, Elements, Edges, intEdges, DbdEdges, NbdEdges, intEdges
- vertices1, vertices2, vertices3,
- Dirichlet, Neumann, EdgeEls, ElementsE.

which can be reached by mesh.Nodes, mesh.Elements and so on, and they are used
by the other functions to form the DG construction. The initial mesh can be uni-
formly refined several times in a "for loop" by calling the function uniformrefine.

for jj = 1:2
mesh = uniformrefine(mesh); %Refine mesh
end
Figure 2: Initial mesh on the unit square $\Omega = [0, 1]^2$ with nodes $n_i$, triangles $E_j$ and edges $e_k$

2.2 User defined quantities

There are certain input values that have to be supplied by the user. Here, we will describe that how one can define these quantities in the main code Main_Linear.m.

One determines the type of the DG method (SIPG, NIPG or IIPG) and the order of the polynomial basis to be used by the variables method and degree, respectively. According to these choices, the values of the penalty parameter and the parameter $\kappa \in \{-1, 1, 0\}$ defining DG method in (3) are set by calling the sub-function set_parameter.

```matlab
% method : NIPG=1, SIPG =2, IIPG=3
method =2;

% Degree of polynomials
degree =1;

% Set up the problem
[penalty, kappa] = set_parameter(method, degree);
```

The next step is to supply the problem parameters. The diffusion constant $\varepsilon$, the advection vector $b$ and the linear reaction term $\alpha$ are defined via the sub-functions fdiff, fadv and freact, respectively.
% Define diffusion, advection, and reaction as subfunctions

% Diffusion
function diff = fdiff(x,y)
    diff = (10^(-6)).*ones(size(x));
end

% Advection
function [adv1, adv2] = fadv(x,y)
    adv1 = (1/sqrt(5))*ones(size(x));
    adv2 = (2/sqrt(5))*ones(size(x));
end

% Linear reaction
function react = freact(x,y)
    react = ones(size(x));
end

The exact solution (if exists) and the source function \( f \) are defined via the subfunctions \( fexact \) and \( fsource \), respectively. Finally, the boundary conditions are supplied via the sub-functions \( DBCexact \) and \( NBCexact \).

% First derivative wrt x
yex_x = (-1./(sqrt(5*diff))).*(sech((2*x-y-0.25)/...)
            (sqrt(5*diff))).^2;

% First derivative wrt y
yex_y = ((0.5)/(sqrt(5*diff))).*(sech((2*x-y-0.25)/...)
            (sqrt(5*diff))).^2;

% Second derivative wrt x
yex_xx = ((0.8)/diff).*tanh((2*x-y-0.25)/(sqrt(5*diff)))/...)
            (sech((2*x-y-0.25)/(sqrt(5*diff)))).^2;

% Second derivative wrt y
yex_yy = ((0.2)/diff).*tanh((2*x-y-0.25)/(sqrt(5*diff)))/...)
            (sech((2*x-y-0.25)/(sqrt(5*diff)))).^2;

% Force function
source = -diff.*(yex_xx+yex_yy)+(adv1.*yex_x+adv2.*yex_y)+...

2.3 Forming and solving linear systems

To form the linear systems, firstly, let us rewrite the discrete DG scheme (3) as

\[
a_h(u_h,v_h) := D_h(u_h,v_h) + C_h(u_h,v_h) + R_h(u_h,v_h) = l_h(v_h) \quad \forall v_h \in V_h, \quad (5)
\]
where the forms $D_h(u_h, v_h)$, $C_h(u_h, v_h)$ and $R_h(u_h, v_h)$ corresponding to the diffusion, convection and linear reaction parts of the problem, respectively

\[
D_h(u_h, v_h) = \sum_{K \in T_h} \int_K \varepsilon \nabla u_h \cdot \nabla v_h \, dx + \sum_{e \in \partial K} \frac{\sigma \varepsilon}{h_e} \int_e [u_h] \cdot [v_h] \, ds - \sum_{e \in \partial K} \int_e \{\varepsilon \nabla u_h \} \cdot [v_h] \, ds + \sum_{e \in \partial K} \int_e \{\varepsilon \nabla v_h \} \cdot [u_h] \, ds
\]

\[
C_h(u_h, v_h) = \sum_{K \in T_h} \int_K \mathbf{b} \cdot \nabla u_h v_h \, dx + \sum_{e \in \partial K} \mathbf{b} \cdot \mathbf{n}(u_h^\text{out} - u_h^\text{in}) v_h \, ds - \sum_{e \in \partial K} \int_e \mathbf{b} \cdot \mathbf{n} u_h^\text{in} v_h \, ds
\]

\[
R_h(u_h, v_h) = \sum_{K \in T_h} \int_K \alpha u_h v_h \, dx + \sum_{e \in \partial K} \int_e g^D \left( \frac{\sigma \varepsilon}{h_e} v_h - \varepsilon \nabla v_h \cdot \mathbf{n} \right) \, ds - \sum_{e \in \partial K} \int_e \mathbf{b} \cdot \mathbf{n} g^D v_h \, ds + \sum_{e \in \partial K} \int_e g^N v_h \, ds,
\]

For a set of basis functions $\{\phi_j\}_{j=1}^N$ spanning the space $V_h$, the discrete solution $u_h \in V_h$ is of the form

\[
u u_h = \sum_{j=1}^N \nu_j \phi_j \quad (7)
\]

where $\nu = (\nu_1, \nu_2, \ldots, \nu_N)^T$ is the unknown coefficient vector. After substituting (7) into (5) and taking $v_h = \phi_i$, we get the linear system of equations

\[
\sum_{j=1}^N \nu_j D_h(\phi_j, \phi_i) + \sum_{j=1}^N \nu_j C_h(\phi_j, \phi_i) + \sum_{j=1}^N \nu_j R_h(\phi_j, \phi_i) = l_h(\phi_i), \quad i = 1, 2, \ldots, N
\]

(8)

Thus, for $i = 1, 2, \ldots, N$, to form the linear system in matrix-vector form, we need the matrices $D, C, R \in \mathbb{R}^{N \times N}$ related to the terms including the forms $D_h, C_h$ and $R_h$ in (8), respectively, satisfying

\[
D\nu + C\nu + R\nu = F
\]

with the unknown coefficient vector $\nu$ and the vector $F \in \mathbb{R}^N$ related to the linear rhs functionals $l_h(\phi_i)$ such that $F_i = l_h(\phi_i), i = 1, 2, \ldots, N$. In the code Main_Keyword, all the matrices $D, C, R$ and the vector $F$ are obtained by calling the function global_system, in which the sub-functions introduced in the previous subsection are used. We set the stiffness matrix, $Stiff$, as the sum of the obtained matrices and we solve the linear system for the unknown coefficient vector $\text{coef} := \nu$. 

8
%Compute global matrices and rhs global vector
[D,C,R,F]=global_system(mesh,@fdiff,@fadv,@freact,...
                     @fsource,@DBCexact,@NBCexact,penalty,kappa,degree);

Stiff=D+C+R; % Stiffness matrix

c coef=Stiff\F; % Solve the linear system

2.4 Plotting the solution

After solving the problem for the unknown coefficient vector, the solutions are
plotted via the the function dg_error, and also the $L^2$-error between the exact and
numerical solution is computed.

% Compute $L^2$-error and plot the solution
[l2err,hmax]=dg_error(coef,mesh,@fexact,@fdiff,degree);

3 Models with non-linear reaction mechanisms

Most of the problems include non-linear source or sink terms. The general model
problem in this case is

$$\alpha u - \varepsilon \Delta u + b \cdot \nabla u + r(u) = f \quad \text{in } \Omega, \quad (9a)$$
$$u = g_D \quad \text{on } \Gamma_D, \quad (9b)$$
$$\varepsilon \nabla u \cdot n = g_N \quad \text{on } \Gamma_N. \quad (9c)$$

which arises from the time discretization of the time-dependent non-linear diffusion-
convection-reaction equations. Here, the coefficient of the linear reaction term,
$\alpha > 0$, stand for the temporal discretization, corresponding to $1/\Delta t$, where $\Delta t$ is the
discrete time-step. The model (9) differs from the model (1) by the additional non-
linear term $r(u)$. To have a unique solution, in addition to the assumptions given
in Section 1, we assume that the non-linear reaction term, $r(u)$, is bounded, locally
Lipschitz continuous and monotone, i.e. satisfies for any $s, s_1, s_2 \geq 0, s, s_1, s_2 \in \mathbb{R}$
the following conditions [7]

$$|r_i(s)| \leq C, \quad C > 0$$
$$\|r_i(s_1) - r_i(s_2)\|_{L^2(\Omega)} \leq L\|s_1 - s_2\|_{L^2(\Omega)}, \quad L > 0$$
$$r_i \in C^1(\mathbb{R}_0^+), \quad r_i(0) = 0, \quad r_i'(s) \geq 0.$$

The non-linear reaction term $r(u)$ occur in chemical engineering usually in the form
of products and rational functions of concentrations, or exponential functions of the
temperature, expressed by the Arrhenius law. Such models describe chemical pro-
cesses and they are strongly coupled as an inaccuracy in one unknown affects all
To solve the non-linear problems, we use the m-file `Main_Nonlinear` which is similar to the m-file `Main_Linear`, but now we use Newton iteration to solve for \( i = 1, 2, \ldots, N \) the non-linear system of equations

\[
\sum_{j=1}^{N} v_j D_h(\phi_j, \phi_i) + \sum_{j=1}^{N} v_j C_h(\phi_j, \phi_i) + \sum_{j=1}^{N} v_j R_h(\phi_j, \phi_i) + \int_{\Omega} r(u_h) \phi_i dx = l_h(\phi_i) \tag{10}
\]

Similar to the linear case, the above system leads to the matrix-vector form

\[
D \boldsymbol{v} + C \boldsymbol{v} + R \boldsymbol{v} + H(\boldsymbol{v}) = F
\]

where, in addition to the matrices \( D, C, R \in \mathbb{R}^{N \times N} \) and the vector \( F \in \mathbb{R}^N \), we also need the vector \( H \in \mathbb{R}^N \) related to the non-linear term such that

\[
H_i(\boldsymbol{v}) = \int_{\Omega} r \left( \sum_{j=1}^{N} v_j \phi_j \right) \phi_i dx, \quad i = 1, 2, \ldots, N.
\]

We solve the nonlinear system by Newton method. For an initial guess \( \boldsymbol{v}^0 = (v_1^0, v_2^0, \ldots, v_N^0)^T \), we solve the system

\[
J^k \boldsymbol{w}^k = -\text{Res}^k \quad (11)
\]

\[
\boldsymbol{v}^{k+1} = \boldsymbol{w}^k + \boldsymbol{v}^k, \quad k = 0, 1, 2, \ldots
\]

until a user defined tolerance is satisfied. In \( \text{Res}^k \) and \( J^k \) denote the vector of system residual and its Jacobian matrix at the current iterate \( \boldsymbol{v}^k \), respectively, given by

\[
\text{Res}^k = (D + C + R) \boldsymbol{v}^k + H(\boldsymbol{v}^k) - F
\]

\[
J^k = D + C + R + HJ(\boldsymbol{v}^k)
\]

where \( HJ(\boldsymbol{v}^k) \) is the Jacobian matrix of the non-linear vector \( H \) at \( \boldsymbol{v}^k \).

\[
HJ(\boldsymbol{v}^k) = \begin{bmatrix}
\frac{\partial H_1(\boldsymbol{v}^k)}{\partial v_1^k} & \frac{\partial H_1(\boldsymbol{v}^k)}{\partial v_2^k} & \cdots & \frac{\partial H_1(\boldsymbol{v}^k)}{\partial v_N^k} \\
\vdots & \ddots & \ddots & \vdots \\
\frac{\partial H_N(\boldsymbol{v}^k)}{\partial v_1^k} & \frac{\partial H_N(\boldsymbol{v}^k)}{\partial v_2^k} & \cdots & \frac{\partial H_N(\boldsymbol{v}^k)}{\partial v_N^k}
\end{bmatrix}
\]

In the code `Main_Nonlinear`, obtaining the matrices \( D, C, R \) and the rhs vector \( F \) is similar to the linear case, but now, additionally, we introduce an initial guess for Newton iteration, and we solve the nonlinear system by Newton method.
% Initial guess for Newton iteration
coef=zeros(size(Stiff,1),1);

% Newton iteration
noi=0;
for ii=1:50
    noi=noi+1;

    % Compute the nonlinear vector and its Jacobian matrix at
    % the current iterate
    [H,HJ]=nonlinear_global(coef,mesh,@freact_nonlinear,degree);

    % Form the residual of the system
    Res = Stiff*coef + H - F;

    % Form the Jacobian matrix of the system
    % (w.r.t. unknown coefficients coef)
    J = Stiff + HJ;

    % Solve the linear system for the correction "w"
    w = J \ (-Res);

    % Update the iterate
    coef = coef + w;

    % Check the accuracy
    if norm(J*w+Res) < 1e-20
        break;
    end
end

To obtain the non-linear vector $H$ and its Jacobian $HJ$ at the current iterate, we call the function nonlinear_global, and it uses the function handle freact_nonlinear which is a sub-function in the file Main_Nonlinear. The sub-function freact_nonlinear has to be supplied by user as the non-linear term $r(u)$ and its derivative $r'(u)$.

% Nonlinear reaction
function [r,dr] = freact_nonlinear(u)
    % Value of the nonlinear reaction term at the current iterate
    r = u.^2;
    % Value of the derivative of the nonlinear reaction
    % term at the current iterate
    dr = 2*u;
end
4 MATLAB routines for main code

Here, we give the main m-file Main_Nonlinearm of the code. The full code is available upon request to the e-mail address uzunca@gmail.com.

1 % This routine solves the diffusion–convection–reaction equation
2 % \alpha u - \epsilon \Delta u + b \cdot \nabla u + r(u) = f
3 % using DG–FEM.
4
5 function Main_Nonlinear()
6
7 clear all
8 clc
9
10 % Generate the mesh
11
12 % Nodes
13 Nodes = [0,0;0.5,0;1,0;0.5,0.5;1,0.5;0,1;0.5,1;1,1];
14 % Elements
15 Elements = [4,1,5;1,2,5;2,6; 2,3,6;7,4,8;4,5,8;5,9,5;6,9];
16 % Dirichlet bdry edges
17 Dirichlet = [1,2;2,3;1,4;3,6;4,7;6,9;7,8;8,9];
18 % Neumann bdry edges
19 Neumann = [];
20 % Initial mesh struct
21 mesh = getmesh(Nodes, Elements, Dirichlet, Neumann);
22
23 for jj=1:2
24 mesh=uniformrefine(mesh); %Refine mesh
25 end
26
27 % method : NIPG=1, SIPG=2, IIPG=3
28 method = 2;
29
30 % Degree of polynomials
31 degree = 1;
32
33 % Set up the problem
34 [penalty, kappa] = set_parameter(method, degree);
35
36 % Compute global matrices and rhs global vector
37 [D,C,R,F] = global_system(mesh, @fdiff, @fadv, @freact, ...
@fs, @DBCexact, @NBCexact, penalty, kappa, degree);

Stiff=D+C+R; % Stiffness matrix

% Initial guess for Newton iteration
coef=zeros(size(Stiff,1),1);

% Newton iteration
noi=0;
for ii=1:50
  noi=noi+1;
  % Compute the nonlinear vector and its Jacobian matrix at
  % the current iterate
  [H,HJ]=nonlinear_global(coef,mesh,@freact_nonlinear,degree);
  % Form the residual of the system
  Res = Stiff*coef + H - F;
  % Form the Jacobian matrix of the system
  % (w.r.t. unknown coefficients coef)
  J = Stiff + HJ;
  % Solve the linear system for the correction "w"
  w = J \ (-Res);
  % Update the iterate
  coef = coef + w;
  % Check the accuracy
  if norm(J*w+Res) < 1e-20
    break;
  end
end

% Compute L2-error and plot the solution
[12err,hmax]=dg_error(coef,mesh,@fexact,@fdiff,degree);

% Degree of freedom
dof=size(mesh.Elements,1)*(degree+1)*(degree+2)*0.5;

fprintf('%DoFs_h_max_L2-error_it\n')
 fprintf(‘%7d,%5.3f,%5.3e,%d\n’,...  
    dof, hmax,l2err, noi);  
end % End of function Main_Nonlinear  

%% Define diffusion, advection, and reaction as subfunctions

% Diffusion
function diff = fdiff(x,y)  
    diff = (10^(-6)).*ones(size(x));  
end

% Advection
function [adv1, adv2] = fadv(x,y)  
    adv1 = (1/sqrt(5))*ones(size(x));  
    adv2 = (2/sqrt(5))*ones(size(x));  
end

% Linear reaction
function react = freact(x,y)  
    react = ones(size(x));
end

% Nonlinear reaction
function [r, dr] = freact_nonlinear(u)  
    r = u.^2;  
    dr = 2*u;
end

%% Define exact solution and force as subfunctions

% Exact solution
function [yex, yex_x, yex_y] = fexact(fdiff,x,y)  
    % Evaluate the diffusion function  
    diff = feval(fdiff,x,y);  
    % Exact value
    yex=0.5*(1-tanh((2*x-y-0.25)./(sqrt(5*diff))));
    % First derivative wrt x  
    yex_x=(-1./(sqrt(5*diff))).*(sech((2*x-y-0.25)./(sqrt(5*diff)))).^2;
    % First derivative wrt y
% Force function
function source = fsource (fdiff, fadv, freact, x, y)
  % Evaluate the diffusion function
  diff = feval (fdiff, x, y);
  % Evaluate the advection function
  [adv1, adv2] = feval (fadv, x, y);
  % Evaluate the linear reaction function
  reac = feval (freact, x, y);
  % Exact value
  yex = 0.5*( 1 - tanh ((2*x-y-0.25)./(sqrt (5*diff))));
  % First derivative wrt x
  yex_x = -1./(sqrt (5*diff)).*(sech ((2*x-y-0.25)./...
            (sqrt (5*diff)))).^2;
  % First derivative wrt y
  yex_y = ((0.5)./(sqrt (5*diff))).*(sech ((2*x-y-0.25)./...
            (sqrt (5*diff)))).^2;
  % Second derivative wrt x
  yex_xx = ((0.8)./diff).*tanh ((2*x-y-0.25)./(...
            (sech ((2*x-y-0.25)./sqrt (5*diff)))).^2;
  % Second derivative wrt y
  yex_yy = ((0.2)./diff).*tanh ((2*x-y-0.25)./(...
            (sech ((2*x-y-0.25)./sqrt (5*diff)))).^2;
  % Force function
  source = -diff.*(yex_xx+yex_yy)+(adv1.*yex_x+adv2.*yex_y)+... reac.*yex+yex.^2;
end

% Boundary Conditions

% D r i c h l e t B o u n d a r y C o n d i t i o n
function DBC=DBCexact (fdiff, x, y)
  % Evaluate the diffusion function
  diff = feval (fdiff, x, y);
  % D r i c h l e t B o u n d a r y C o n d i t i o n
  DBC=0.5*(1-tanh ((2*x-y-0.25)./sqrt (5*diff))));
end
function NC = NBCexact(mesh, fdiff, x, y)

function [penalty, kappa] = set_parameter(method, degree)

References

[1] D. Arnold, F. Brezzi, B. Cockburn, and L. Marini: Unified analysis of discontinuous Galerkin methods for elliptic problems. SIAM J. Numer. Anal., 39, 1749-1779, (2002).
[2] B. Ayuso, and L.D. Marini: Discontinuous Galerkin methods for advection-diffusion-reaction problems. *SIAM J. Numer. Anal.*, 47, 1391-1420, (2009).

[3] L. Chen: *iFEM: an innovative finite element method package in MATLAB*, an innovative finite element methods package in MATLAB. *Tech. rep.: Department of Mathematics, University of California, Irvine*, (2008).

[4] P. Houston, C. Schwab, and E. Süli: Discontinuous hp-finite element methods for advection-diffusion-reaction problems. *SIAM J. Numer. Anal.*, 39, 2133-2163, (2002).

[5] P. d. Leva: MULTIPROD TOOLBOX, Multiple matrix multiplications, with array expansion enabled, University of Rome Foro Italico, Rome.

[6] B. Rivière: *Discontinuous Galerkin methods for solving elliptic and parabolic equations. Theory and implementation*, SIAM, (2008).

[7] M. Uzunca, B. Karasözen, and M. Manguoğlu: Adaptive discontinuous Galerkin methods for non-linear diffusion-convection-reaction equations. *Computers and Chemical Engineering*, 68, 24-37, (2014).

[8] R. Verfürth: *A posteriori Error Estimates Techniques for Finite Element Methods*. Oxford University Press, (2013).