A MatLab Tutorial for Diffusion-Convection-Reaction Equations using dGFEM

Murat Uzunca
Department of Mathematics, Sinop University, 57000 Sinop, Turkey
muzunca@sinop.edu.tr

Bülent Karasözen
Institute of Applied Mathematics & Department of Mathematics
Middle East Technical University, 06800 Ankara, Turkey
bulent@metu.edu.tr

Abstract:
We discuss a collection of MatLab routines using discontinuous Galerkin (dG) methods for solving and simulating steady-state diffusion-convection-reaction equations in 2D. The code employs the sparse matrix facilities of MatLab with the coding style ”vectorization” which replaces for loops by matrix operations. Moreover, we utilize multiple matrix multiplications *”MULTIPROD”* [6] to decrease the number of for loops in an efficient way.

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 (linear) model problem used in the code is

$$\alpha u - \varepsilon \Delta u + \mathbf{b} \cdot \nabla u = f \quad \text{in } \Omega, \tag{1a}$$

$$u = g^D \quad \text{on } \Gamma^D, \tag{1b}$$

$$\varepsilon \nabla u \cdot \mathbf{n} = g^N \quad \text{on } \Gamma^N. \tag{1c}$$

The domain Ω 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, $\mathbf{b} \in \left(W^{1,\infty}(\Omega)\right)^2$ 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 $\mathbf{n}$ denote the unit outward normal vector to the boundary.

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

$$\int_\Omega (\varepsilon \nabla u \cdot \nabla v + \mathbf{b} \cdot \nabla u v + \alpha u v) dx = \int_\Omega f v dx + \int_{\Gamma^N} g^N v ds\,, \quad \forall v \in V \tag{2}$$

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

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

The next step of the classical (continuous) FEM is to find an approximation to the problem (2) using a conforming, finite-dimensional subspace $V_h \subset V$, which requires that the space V_h contains functions of particular smoothness (e.g. when $V = H_0^1(\Omega)$, then we choose $V_h \subset \{v \in C(\overline{\Omega}) : v = 0 \text{ on } \partial\Omega\}$). On the other hands, discontinuous Galerkin methods make it easy to use the non-conforming spaces, in which case the functions in $V_h \not\subset V$ are allowed to be discontinuous on the inter-element boundaries.

In our code, the discretization of the problem (1) is based on the discontinuous Galerkin methods for the diffusion part [1, 7] and the upwinding for the convection part [2, 5]. Let $\{\xi_h\}$ be a family of shape regular meshes with the elements (triangles) $K_i \in \xi_h$ satisfying $\overline{\Omega} = \cup \overline{K}$ and $K_i \cap K_j = \emptyset$ for K_i, $K_j \in \xi_h$. Let us denote by Γ_0, Γ_D and Γ_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 $\mathbb{P}_k(K)$ be the set of all polynomials of degree at most k on K. Then, set the finite dimensional solution and test function space by

$$V_h = \left\{v \in L^2(\Omega) : v|_K \in \mathbb{P}_k(K),\ \forall K \in \xi_h\right\} \not\subset V.$$

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, $K_j \in \xi_h$ $(i < j)$ 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

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

where $\mathbf{n}_e$ 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

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

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

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

where $\mathbf{n}$ 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

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

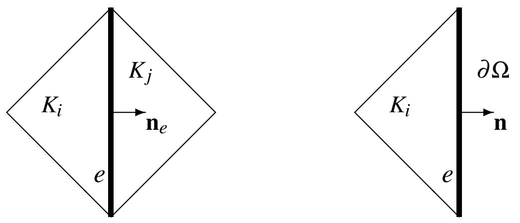

Figure 1: Two adjacent elements sharing an edge (left); an element near to domain boundary (right)

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

$$a_h(u_h, v_h) = l_h(v_h) \qquad \forall v_h \in V_h, \tag{3}$$

$$\begin{aligned}
a_h(u_h,v_h) =& \sum_{K\in\xi_h} \int_K \varepsilon \nabla u_h \cdot \nabla v_h dx + \sum_{K\in\xi_h} \int_K (\mathbf{b}\cdot\nabla u_h + \alpha u_h) v_h dx \\
&- \sum_{e\in\Gamma_0\cup\Gamma_D} \int_e \{\varepsilon\nabla u_h\}\cdot[v_h] ds + \kappa \sum_{e\in\Gamma_0\cup\Gamma_D} \int_e \{\varepsilon\nabla v_h\}\cdot[u_h] ds \\
&+ \sum_{K\in\xi_h} \int_{\partial K^- \setminus \partial\Omega} \mathbf{b}\cdot\mathbf{n}(u_h^{out} - u_h^{in}) v_h ds - \sum_{K\in\xi_h} \int_{\partial K^- \cap \Gamma^-} \mathbf{b}\cdot\mathbf{n} u_h^{in} v_h ds \\
&+ \sum_{e\in\Gamma_0\cup\Gamma_D} \frac{\sigma\varepsilon}{h_e} \int_e [u_h]\cdot[v_h] ds, \\
l_h(v_h) =& \sum_{K\in\xi_h} \int_K f v_h dx + \sum_{e\in\Gamma_D} \int_e g^D \left(\frac{\sigma\varepsilon}{h_e} v_h - \varepsilon\nabla v_h \cdot \mathbf{n} \right) ds \\
&- \sum_{K\in\xi_h} \int_{\partial K^- \cap \Gamma^-} \mathbf{b}\cdot\mathbf{n} g^D v_h ds + \sum_{e\in\Gamma_N} \int_e g^N v_h ds,
\end{aligned}$$

where u_h^{out} and u_h^{in} denotes the values on an edge from outside and inside of an element K, respectively. The parameter κ 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 $\sigma \in \mathbb{R}_0^+$ is called the penalty parameter which should be sufficiently large; independent of the mesh size h and the diffusion coefficient ε [7] [Sec. 2.7.1]. In our code, we choose the penalty parameter σ 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 line-by-line descriptions of our main code. The use of the code consists of three main 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, in the case of our code, 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*

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 [4] 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_Linear.m* through the lines 15–21, and we call the *getmesh* function to form the initial mesh structure *mesh*, line 23.

```
% Generate the mesh

% Nodes
Nodes = [0,0;0.5,0;1,0;0,0.5;0.5,0.5;1,0.5;0,1;0.5,1;1,1];
% Elements
Elements = [4,1,5;1,2,5;5,2,6; 2,3,6;7,4,8;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 element 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 **Neumann** 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:

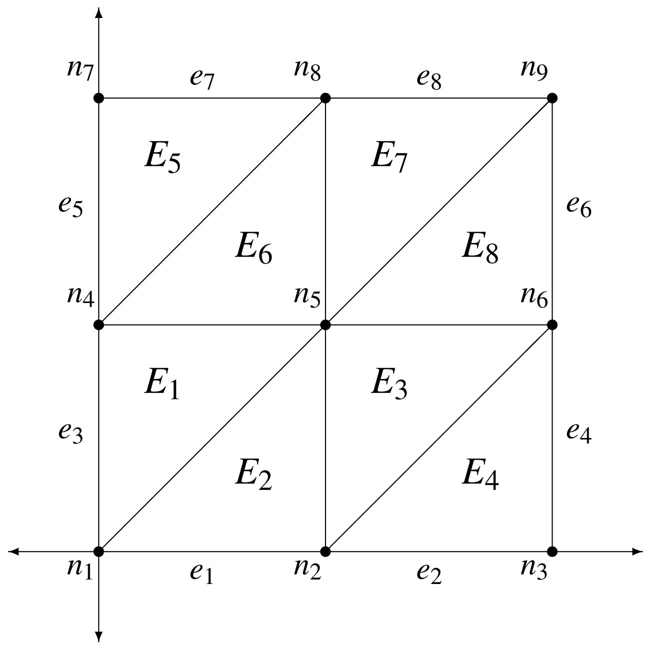

Figure 2: Initial mesh on the unit square $\Omega = [0,1]^2$ with nodes n_i, triangles E_j and edges e_k

- 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. In line 25–27, the initial mesh is uniformly refined several times in a ”for loop” by calling the function *uniformrefine*.

```
for jj=1:2
    mesh=uniformrefine(mesh);  %Refine mesh
end
```

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*.

In lines 29–33, 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* in line 36.

```
% 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. In line 59–75, the diffusion constant ε, the advection vector $\mathbf{b}$ and the linear reaction term α 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 in lines 77–101 via the sub-functions *fexact* and *fsource*, respectively. Finally, in lines 104–118, the boundary conditions are supplied via the sub-functions *DBCexact* and *NBCexact*.

```
%% Boundary Conditions

% Drichlet Boundary Condition
function DBC=DBCexact(fdiff,x,y)
    % Evaluate the diffusion function
    diff  = feval(fdiff,x,y);
    %Drichlet Boundary Condition
    DBC=0.5*(1-tanh((2*x-y-0.25)./(sqrt(5*diff))));
end

% Neumann Boundary Condition
function NC = NBCexact(mesh,fdiff,x,y)
    %Neumann Boundary Condition
    NC=zeros(size(x));
end
```

2.3 Forming and solving linear systems

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

$$a_h(u_h,v_h) := D_h(u_h,v_h) + C_h(u_h,v_h) + R_h(u_h,v_h) = l_h(v_h) \qquad \forall v_h \in V_h, \tag{5}$$

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

$$\begin{aligned}
D_h(u_h, v_h) &= \sum_{K\in\xi_h} \int_K \varepsilon \nabla u_h \cdot \nabla v_h dx + \sum_{e\in\Gamma_0\cup\Gamma_D} \frac{\sigma\varepsilon}{h_e} \int_e [u_h]\cdot[v_h] ds \\
&\quad - \sum_{e\in\Gamma_0\cup\Gamma_D} \int_e \{\varepsilon\nabla u_h\}\cdot[v_h] ds + \kappa \sum_{e\in\Gamma_0\cup\Gamma_D} \int_e \{\varepsilon\nabla v_h\}\cdot[u_h] ds \\
C_h(u_h, v_h) &= \sum_{K\in\xi_h} \int_K \mathbf{b}\cdot\nabla u_h v_h dx \\
&\quad + \sum_{K\in\xi_h} \int_{\partial K^-\setminus\partial\Omega} \mathbf{b}\cdot\mathbf{n}(u_h^{out} - u_h^{in}) v_h ds - \sum_{K\in\xi_h} \int_{\partial K^-\cap\Gamma^-} \mathbf{b}\cdot\mathbf{n} u_h^{in} v_h ds \\
R_h(u_h, v_h) &= \sum_{K\in\xi_h} \int_K \alpha u_h v_h dx \\
l_h(v_h) &= \sum_{K\in\xi_h} \int_K f v_h dx + \sum_{e\in\Gamma_D} \int_e g^D \left(\frac{\sigma\varepsilon}{h_e} v_h - \varepsilon\nabla v_h \cdot \mathbf{n} \right) ds \\
&\quad - \sum_{K\in\xi_h} \int_{\partial K^-\cap\Gamma^-} \mathbf{b}\cdot\mathbf{n} g^D v_h ds + \sum_{e\in\Gamma_N} \int_e g^N v_h ds,
\end{aligned}$$

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

$$u_h = \sum_{j=1}^N \upsilon_j \phi_j \tag{7}$$

where $\upsilon = (\upsilon_1, \upsilon_2, \ldots, \upsilon_N)^T$ is the unknown coefficient vector. After substituting (7) into (5) and taking $v_h = \phi_i$, we get the linear system of equations

$$\sum_{j=1}^N \upsilon_j D_h(\phi_j, \phi_i) + \sum_{j=1}^N \upsilon_j C_h(\phi_j, \phi_i) + \sum_{j=1}^N \upsilon_j R_h(\phi_j, \phi_i) = l_h(\phi_i), \quad i = 1, 2, \ldots, N \tag{8}$$

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

$$D\upsilon + C\upsilon + R\upsilon = F$$

with the unknown coefficient vector υ 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_Linear.m*, all the matrices D, C, R and the vector F are obtained by calling the function *global_system* in lines 38–40, in which the sub-functions introduced in the previous subsection are used. In line 42, we set the stiffness matrix, *Stiff*, as the sum of the obtained matrices and we solve the linear system in line 44 for the unknown coefficient vector *coef*:= υ.

```
%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

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* in line 47, and also the L^2-error between the exact and numerical solution is computed.

```
% Compute L2-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

$$\alpha u - \varepsilon \Delta u + \mathbf{b} \cdot \nabla u + r(u) = f \quad \text{in } \Omega, \tag{9a}$$
$$u = g^D \quad \text{on } \Gamma^D, \tag{9b}$$
$$\varepsilon \nabla u \cdot \mathbf{n} = g^N \quad \text{on } \Gamma^N. \tag{9c}$$

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 Δt 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 [3]

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

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 processes and they are strongly coupled as an inaccuracy in one unknown affects all the others.

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 $i = 1, 2, \dots, N$ the non-linear system of equations

$$\sum_{j=1}^{N} \upsilon_j D_h(\phi_j, \phi_i) + \sum_{j=1}^{N} \upsilon_j C_h(\phi_j, \phi_i) + \sum_{j=1}^{N} \upsilon_j R_h(\phi_j, \phi_i) + \int_\Omega r(u_h)\phi_i dx = l_h(\phi_i) \qquad (10)$$

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

$$D\upsilon + C\upsilon + R\upsilon + H(\upsilon) = F$$

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

$$H_i(\upsilon) = \int_\Omega r\left(\sum_{j=1}^{N} \upsilon_j \phi_j\right)\phi_i dx\,, \quad i = 1, 2, \dots, N.$$

We solve the nonlinear system by Newton method. For an initial guess $\upsilon^0 = (\upsilon_1^0, \upsilon_2^0, \dots, \upsilon_N^0)^T$, we solve the system

$$\begin{aligned} J^k w^k &= -Res^k \\ \upsilon^{k+1} &= w^k + \upsilon^k\,, \quad k = 0, 1, 2, \dots \end{aligned} \qquad (11)$$

until a user defined tolerance is satisfied. In (11), Res^k and J^k denote the vector of system residual and its Jacobian matrix at the current iterate υ^k, respectively, given by

$$\begin{aligned} Res^k &= S\upsilon^k + H(\upsilon^k) - F \\ J^k &= S + HJ(\upsilon^k) \end{aligned}$$

where $HJ(\upsilon^k)$ is the Jacobian matrix of the non-linear vector H at υ^k

$$HJ(\upsilon^k) = \begin{bmatrix} \frac{\partial H_1(\upsilon^k)}{\partial \upsilon_1^k} & \frac{\partial H_1(\upsilon^k)}{\partial \upsilon_2^k} & \cdots & \frac{\partial H_1(\upsilon^k)}{\partial \upsilon_N^k} \\ \vdots & \ddots & & \vdots \\ \frac{\partial H_N(\upsilon^k)}{\partial \upsilon_1^k} & \frac{\partial H_N(\upsilon^k)}{\partial \upsilon_2^k} & \cdots & \frac{\partial H_N(\upsilon^k)}{\partial \upsilon_N^k} \end{bmatrix}$$

In the code *Main_Nonlinear*, obtaining the matrices D, C, R and the rhs vector F is similar to the linear case. In line 45, we initialize the initial guess for Newton iteration, and we solve the nonlinear system in lines 47–74.

```
% 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* in line 54, and it uses the function handle *freact_nonlinear* which is a sub-function in the file *Main_Nonlinear*, lines 106–114. 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
```

References

[1] D. Arnold, F. Brezzi, B. Cockborn, 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] M. Bause, and K. Schwegler: Analysis of stabilized higher-order finite element approximation of nonstationary and non-linear convection-diffusion-reaction equations. *Comput. Methods Appl. Mech. Engrg.*, **209-212**, 184-196, (2012).

[4] L. Chen: *i*FEM: 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).

[5] 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).

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

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

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