Finite Elements for Fluids - Coding

Arthur Lustman

February 19, 2019

This report for the Finite Element for Fluids class is split in two sections : the implementation of the different part of the code and the redaction of the exercise.

1 Implementation of the code

This section describe the reasoning behind the implementation of parts of the code : the quadratic polynomials, the source term and the completion of the SUPG and GLS methods.

1.1 Quadratic polynomials

The code works very well by itself, the only lines of code that need to be added are the one to obtain the coordinates of the nodes `X` and the connectivity matrix `T`.

```
nPt = nElem*p + 1;
h = (dom(2) - dom(1))/(nPt-1);
X = (dom(1):h:dom(2))';
if p==1
    T = [1:nPt-1; 2:nPt]';
else
    T = [1:p:nPt-1; 2:p:nPt; 3:p:nPt+1]';
end
```

The algorithm generates by itself the quadratic elements and the shape functions associated because of the line of code.

```
referenceElement = SetRefereceElement(p);
```

The problem is that the algorithm doesn't know how to plot the quadratic functions. It is used to join the line linearly between each nodes, which is not enough here. A function `plot_quad` is added to work this out. The function used the values of the solution computed `sol`, the Dirichlet boundary conditions `ValDir`, the coordinates of the nodes `X` and the number of elements `nElem` to plot a quadratic solution. The entire domain is split between elements and divided into further value to plot a continuous looking solution. A quadratic polynomial is evaluated with the coordinates of the 3 nodes of the element (using `polyfit` and `polyval`) and added to the list of coordinates `x0` and `y0`.

```
function [x0,y0]=plot_quad(sol,X,nElem,ValDir)
     x0 = [];
     y0 = [];
for i = 1 : nElem
     dom = X(2*i-1:2*i+1)';
     if i==1
          prec = [ValDir(1);sol(1:2)]';
     elseif i==nElem
          prec = [sol(end-1:end);ValDir(2)]';
     else
          prec = sol(2*i-2:2*i)';
     end

     p = polyfit(dom,prec,2);
     first = linspace(dom(1),dom(3),40);
     second = polyval(p,first);

     x0 = [x0 first(1:end-1)];
     y0 = [y0 second(1:end-1)];
end

x0 = [x0 X(end)];
y0 = [y0 ValDir(2)];
```

The result is pictured in the following figure : a 1D convection-diffusion equation with a source term solved using the Galerkin methods and 10 quadratic elements.

$$\boldsymbol{a}\cdot\nabla u - \nabla\cdot(\nu\nabla u) + \sigma u = s(x) \qquad \text{in } \Omega \tag{1}$$

$$u = u_D \qquad \text{on } \partial\Omega \tag{2}$$

$$s(x) = 10\exp(-5x) - 4\exp(-x) \tag{3}$$

The convection-diffusion constants are $a = 1$, $\nu = 0.01$, $\sigma = 0$ in a domain $\Omega = [0,1]$ with the Dirichlet conditions $u(0) = 0$ and $u(1) = 1$

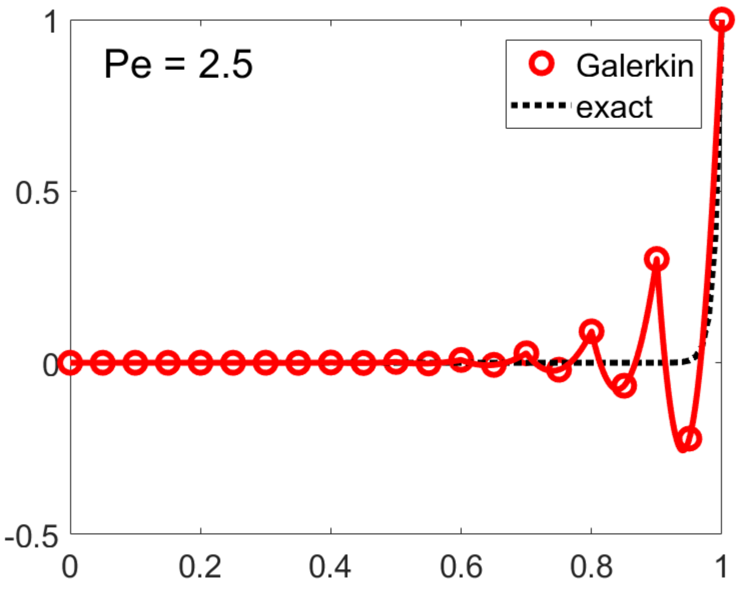

Figure 1: Galerkin's method for quadratic elements

1.2 The Source term

Implementing or modifying the source term is simply done by modifying a couple of lines in two Matlab function files `SourceTerm` which is used to compute the solution and in `ExactSol` which is used to plot the analytic solution.

I implemented the source term proposed in the exercise

$$s(x) = 10\exp(-5x) - 4\exp(-x)$$

In order to reach this source term a 4th problem is added and follows the pattern as the previous problem

SourcTerm

```
elseif problem == 5
    res = 10*exp(-5*x)-4*exp(-x);
```

The analytic solution is then added to the other function

ExactSol

```
elseif problem == 4
    aux = 5*nu^2 + 6*a*nu + a^2;
    res = ( exp(a/nu)*(-18*nu-2*a)+exp(a*x/nu)*(5*nu^2+...
```

The result of the problem number 4 can be seen on the previous figure 1.

1.3 Resolution methods

Both method needs a separate Matlab file which works the same way. The only difference comes from the computation of the elemental stiffness $\boldsymbol{Ke}$ and force vector $\boldsymbol{fe}$.

The following work is based on the integrals developed in the slides or the book of reference. Since the exercise is in 1D, we are working in such dimension and adapting the integrals.

The first work is performed by changing the stabilization parameter: the usage of quadratic elements demands two different stabilization parameter, for the external nodes and for the mid-side node of the element. This is regrouped in the variable `MTau` which is either a scalar or a matrix of the following form

$$\begin{bmatrix} \texttt{tau\_c} & 0 & 0 \\ 0 & \texttt{tau} & 0 \\ 0 & 0 & \texttt{tau\_c} \end{bmatrix} \tag{4}$$

Where the values of `tau` and `tau_c` are differently computed[1]. This is implemented with the following lines of code

```
    if length(N)==3
        tau_c = example.tau_c;
        MTau = eye(3).*[tau_c; tau; tau_c];
    else
        MTau = tau;
    end
```

[1]Jean Donea and Antonio Huerta, *Finite Element Methods for Flow Problems*, West Sussex: Wiley, 2003. Page 56 and 62

1.3.1 Streamline Upwind Petrov-Galerkin (SUPG)

$$\int_\Omega (w(a\partial_x u) + \partial_x w \cdot (\nu \partial_x u) + w\sigma u)\, d\Omega + \sum_e \int_{\Omega_e} (a\partial_x w)\boldsymbol{\tau} \left((a\partial_x u) - \partial_x(\nu\partial_x u) + \sigma u\right) d\Omega \tag{5}$$

$$= \int_\Omega ws\, d\Omega + \sum_e \int_{\Omega_e} (a\partial_x w)\tau s\, \Omega \tag{6}$$

A few term needs to be changed, performing discretization

$$u(x) = \sum u_x N(x) \tag{7}$$

$$w(x) = \sum w_x N(x) \tag{8}$$

For which the derivatives are computed

$$\partial_x u(x) = \sum u_x \partial_x N(x) \tag{9}$$

$$\partial_{xx} u(x) = \sum u_x \partial_{xx} N(x) \tag{10}$$

Which gives us the following result as lines of code

SUPG

```
for ig = 1:ngaus
     [...]
     Nxx_ig = N2xi(ig,:)*(2/h)^2;
     [...]
     Ke = Ke + w_ig*(N_ig'*a*Nx_ig + Nx_ig'*nu*Nx_ig) ...
          + (w_ig*sigma*N_ig)'*N_ig ...
          + w_ig*MTau*(a*Nx_ig)'*(a*Nx_ig-nu*Nxx_ig+sigma*N_ig);
     [...]
     fe = fe + w_ig*(N_ig)'*s + w_ig*MTau*(a*Nx_ig)'*s;
end
```

1.3.2 Galerkin least-squares (GLS)

The work previously stated has to be done here too

$$\int_\Omega (w(a\partial_x u) + \partial_x w \cdot (\nu \partial_x u) + w\sigma u)\, d\Omega + \sum_e \int_{\Omega_e} (a\partial_x w - \partial_x(\nu w) + \sigma w)\tau \left((a\partial_x u) - \partial_x(\nu\partial_x u) + \sigma u\right) d\Omega \tag{11}$$

$$= \int_\Omega ws\, d\Omega + \sum_e \int_{\Omega_e} (a\partial_x w - \partial_x(\nu\partial_x w) + \sigma w)\tau s\, \Omega \tag{12}$$

Based on the previous changes already done to the SUPG Matlab code, only the elemental stiffness matrix and the forcing vector are to be changed after discretization.

GLS

```
     Ke = Ke + w_ig*(N_ig'*a*Nx_ig + Nx_ig'*nu*Nx_ig) ...
          + (w_ig*sigma*N_ig)'*N_ig ...
          + w_ig*MTau*(a*Nx_ig - nu*Nxx_ig + sigma*N_ig)'* ...
          (a*Nx_ig - nu*Nxx_ig + sigma*N_ig);
     [...]
     fe = fe + w_ig*(N_ig)'*s + MTau*w_ig*(a*Nx_ig - ...
     nu*Nxx_ig + sigma*N_ig)'*s;
```

2 Homework

This section of the report is dedicated to compute the instructions in the slide and observe the results based on the code already explained in the previous section.

2.1 Galerkin's method

The problem's statement is the first one, where the source term is equal to 0 everywhere and the boundary conditions are 0 on the left and 1 on the right. Only linear elements are used in this subsection.

Changing the two parameters a and ν may change the shape of the solution, as is can be seen by the difference of the exact solution in figure 2 to figure 3, 4 and 5. Figures 3, 4 and 5 display the same exact solution because the ration a/ν is the same, which is much different for the case of figure 2.

Among figure 3, 4 and 5, it is the last one that display the best computed solution as the number of element is increased. Thus the Peclet number is decreased which result in a non oscillating computed solution. Though it can be seen that the computation is not exact due to the boundary layer on the right side of the solution.

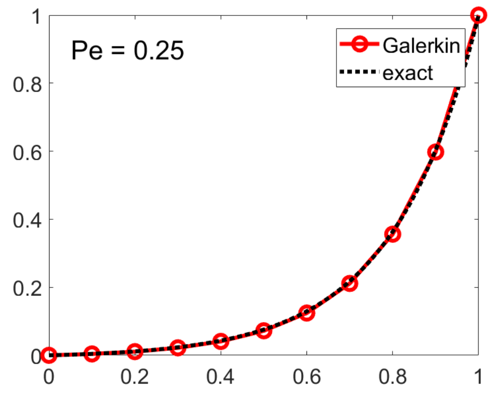

Figure 2: a = 1, ν = 0.2, 10 elements

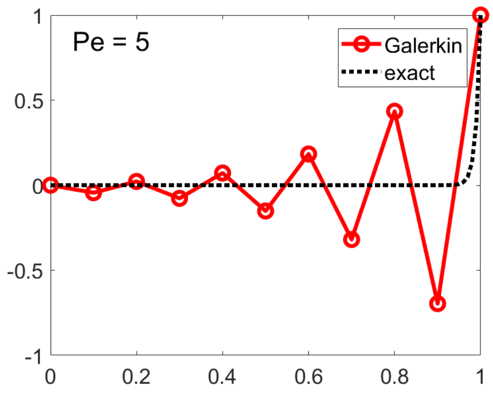

Figure 3: a = 20, ν = 0.2, 10 elements

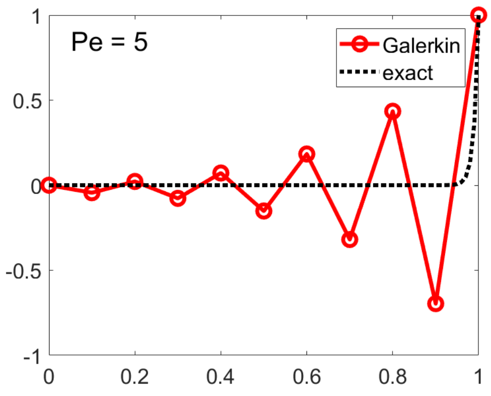

Figure 4: a = 1, ν = 0.01, 10 elements

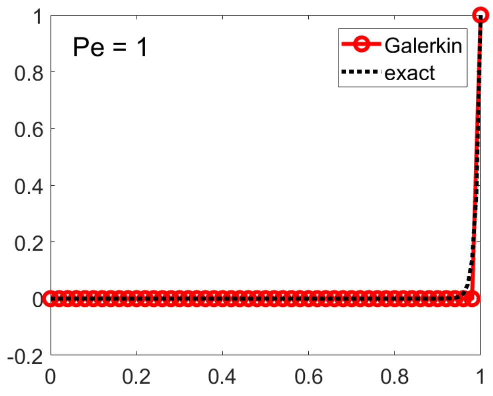

Figure 5: a = 1, ν = 0.01, 50 elements

2.2 Default parameters with different methods

The third case (a = 1, ν = 0.01, 10 linear elements), renamed as the default parameters, gives us interesting results when the different solution methods are computed and compared.

Galerkin's method is the only one offering an oscillating solution. The other methods, displayed in figures 7, 8 and 9 do the offer exact solution at the node. They are of course incapable to display the exact solution near the right boundary layer due to the linear nature of the elements and their small amount used on the domain.

The optimal stabilization parameter is the same for every method and is equal to $\tau = 0.040005$. Since linear elements are used, it is not possible to use a corner stabilization term.

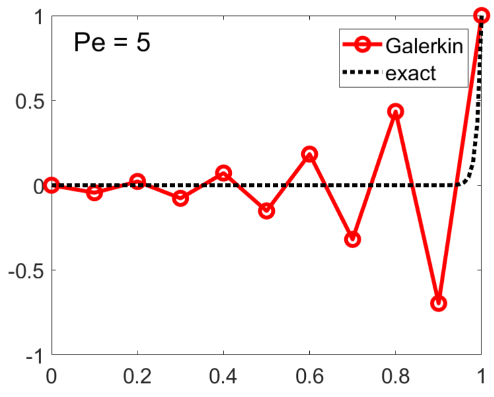

Figure 6: Galerkin's method

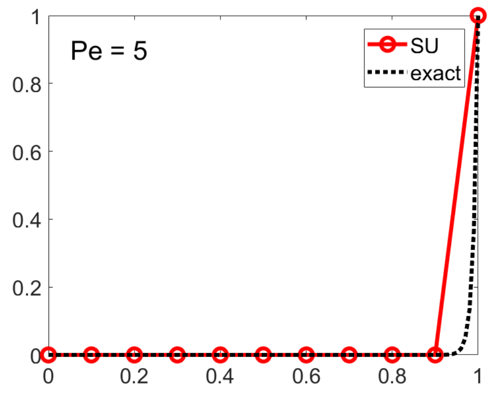

Figure 7: Streamline Upwind method

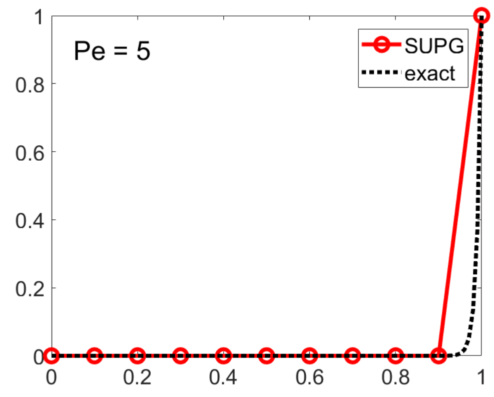

Figure 8: SUPG's method

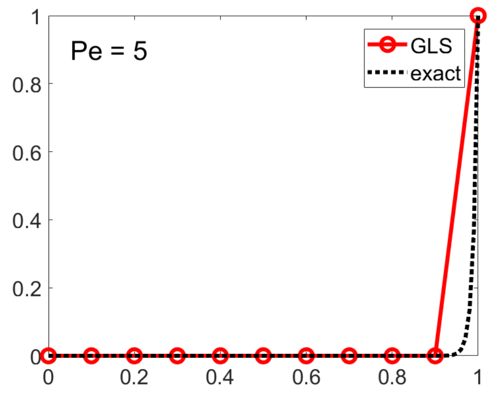

Figure 9: Galerkin least-squares method

2.3 Changed source term

The source term is changed, it's implementation is described in section 1.2, and doing so modify the shape of the solution of the problem.

One again, Galerkin method doesn't do well and oscillate in the whole domain. The previous exact solution's computed for the 3 other methods are not exact here. It is a result of the solution being more complicated than before.

The SU method's display a computed solution that is under the exact solution but still has the same type or curvature. The SUPG and GLS solutions are the closest to the solutions but they still show a small difference between the computation and the exact solution.

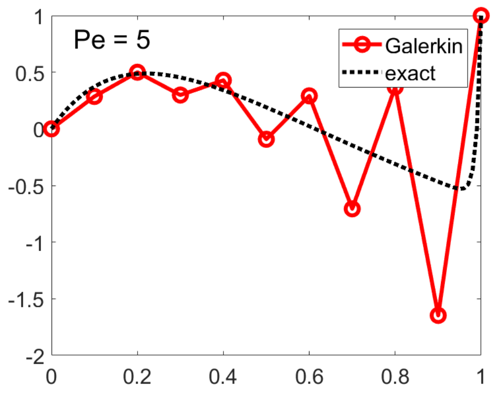

Figure 10: Galerkin's method

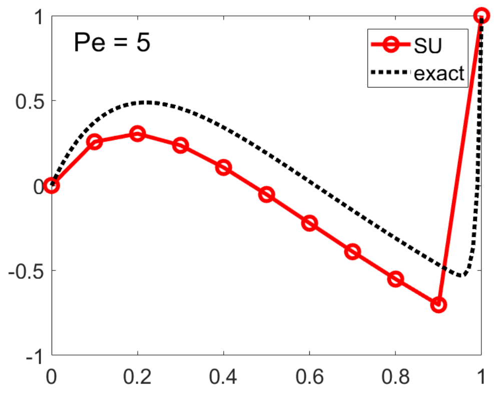

Figure 11: Streamline Upwind method

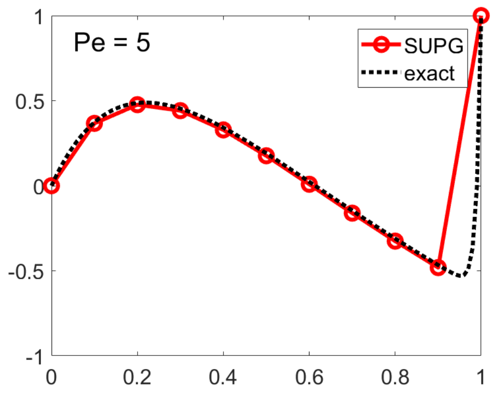

Figure 12: SUPG's method

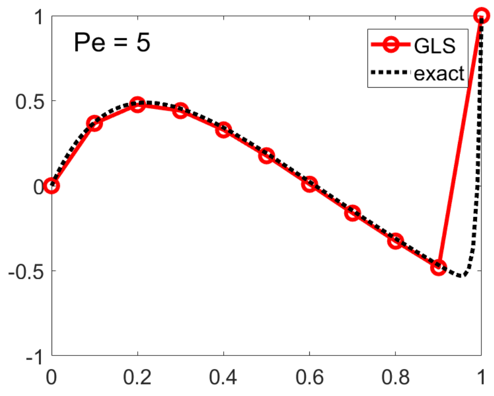

Figure 13: Galerkin least-squares method

2.4 Quadratic elements

The previous methods for the first source term displayed exact solution at the node (except for Galerkin) but had trouble with the right boundary layer. The Galerkin's method has improved for quadratic elements, as it can be seen on figure 1 for the first problem type with the default parameters. Oscillation still appears but is located near the right boundary layer.

The solution of the other methods which were already exact at the node with linear elements did not improved with quadratic elements due to the right boundary layer. The computation methods would display better solutions if the number of elements is refined.

The default parameters are used for the second source term (equation 3) and the result of the computed solution are displayed on figures 14 to 17. The oscillating behavior of Galerkin's method is once again uncovered here, close to the right boundary layer, but it has greatly improved from the linear elements. All methods behaves bad close to the boundary layer but SUPG's nodes seem to be more precise than the other. Although really close, the solution of Galerkin, SUPG and GLS, never have an exact solution at the nodes.

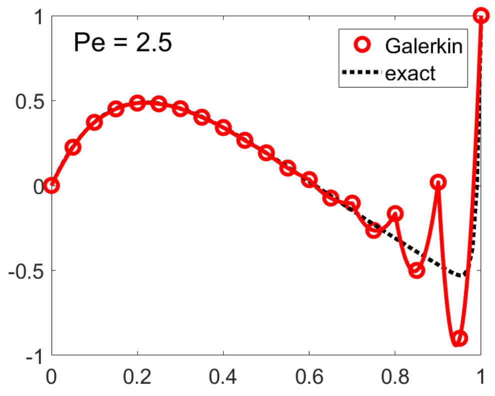

Figure 14: Galerkin's method

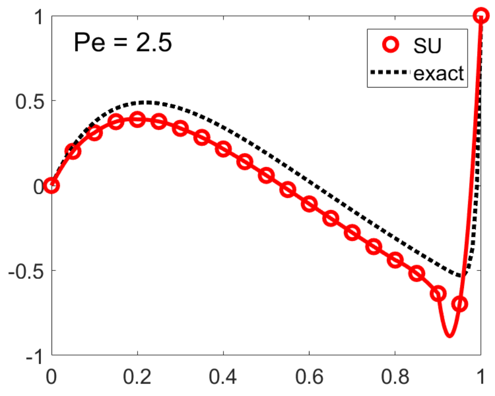

Figure 15: Streamline Upwind method

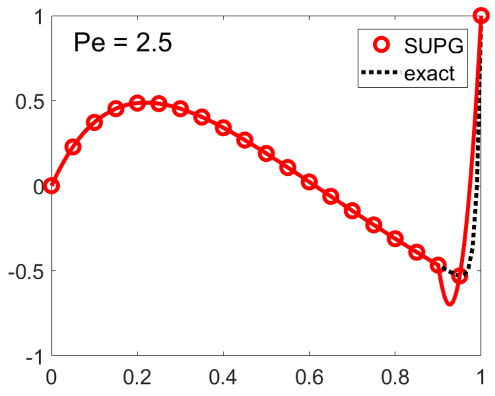

Figure 16: SUPG's method

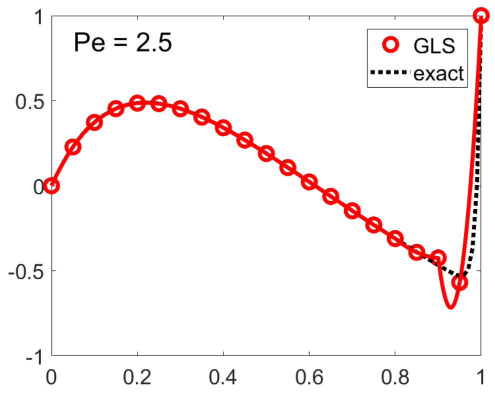

Figure 17: Galerkin least-squares method