ENM 502 : ASSIGNMENT #5

Submitted on 13th May 2014 by :

Ramalingam Kailasham

kailr@seas.upenn.edu

(b) Effect of variation of parameters on the solution

Changing "k" while holding D and v constant

As the rate constant is increased while keeping the diffusivity and the convection term constant, we see that the concentration drops to zero before x=1. This tallies with the expected result.

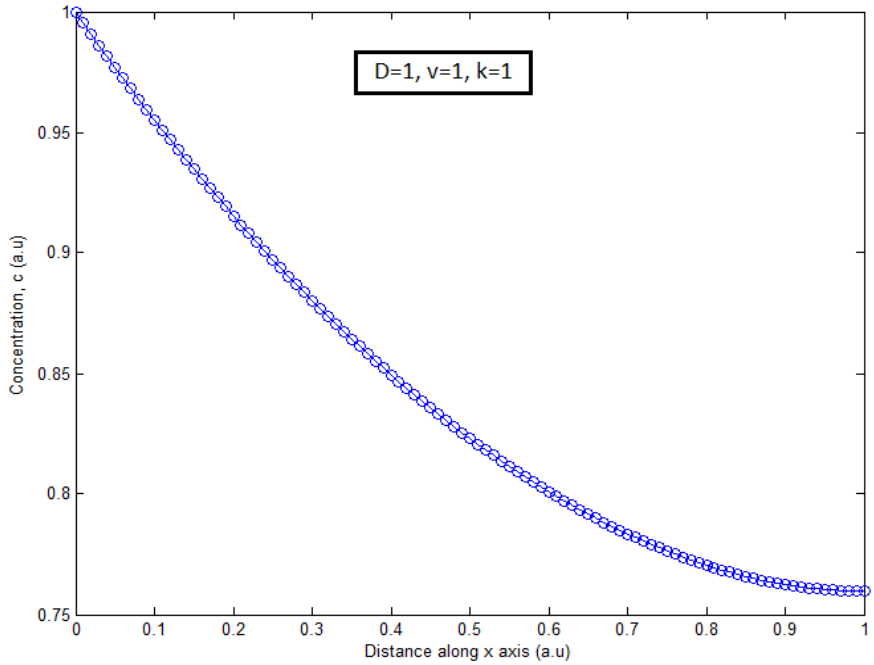

Figure 1 : Plot showing variation of c with x for D=1,v=1,k=1

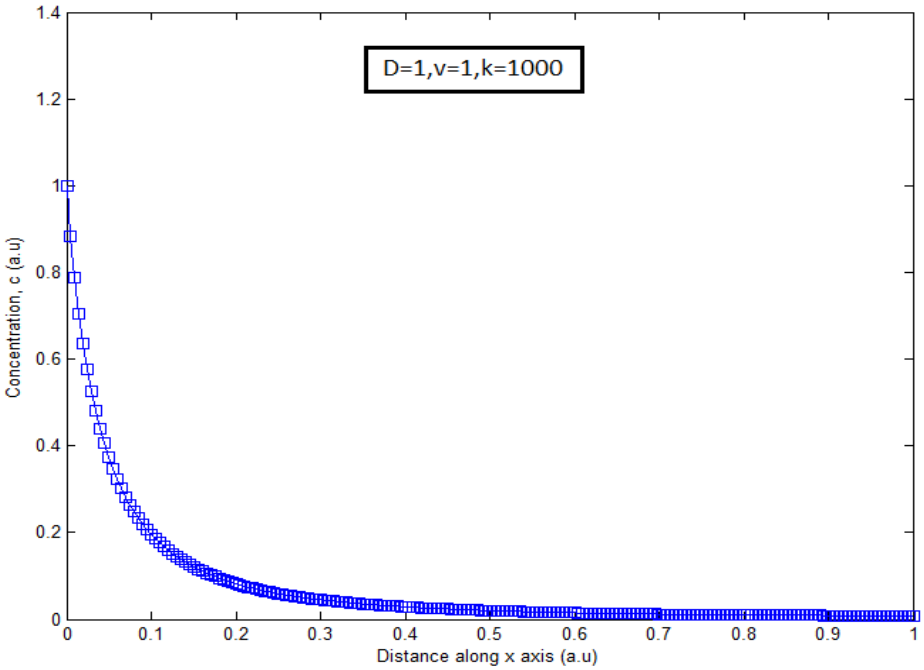

Figure 2 : Plot showing variation of c with x for D=1,v=1, k=1000

Changing "v" while holding D and k constant

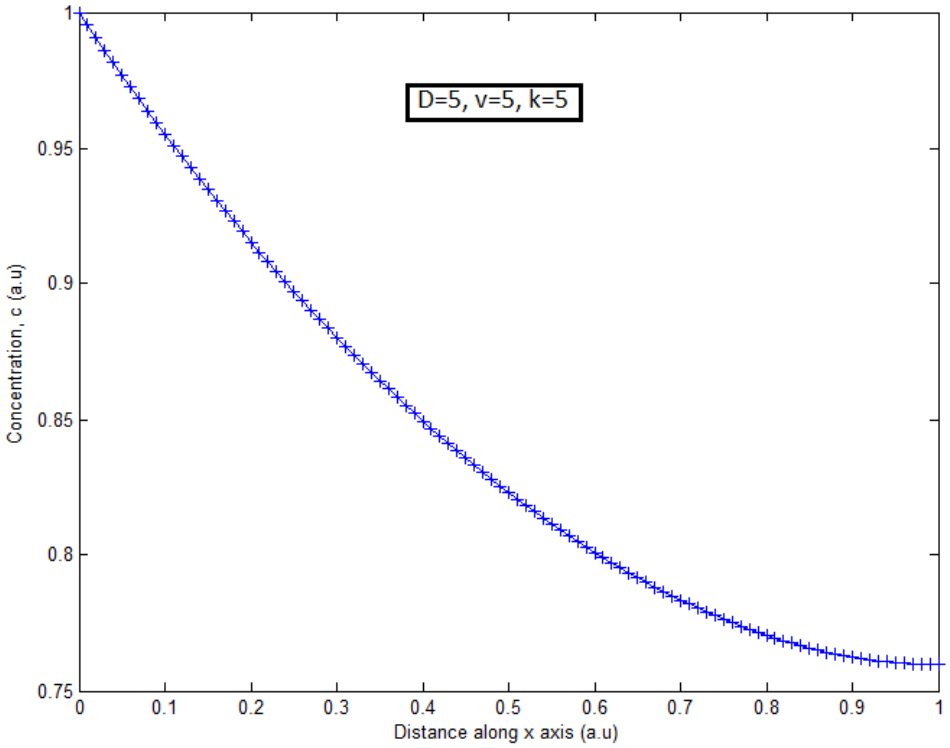

Figure 3 : Plot showing variation of c with x for D=5, v=5, k=5

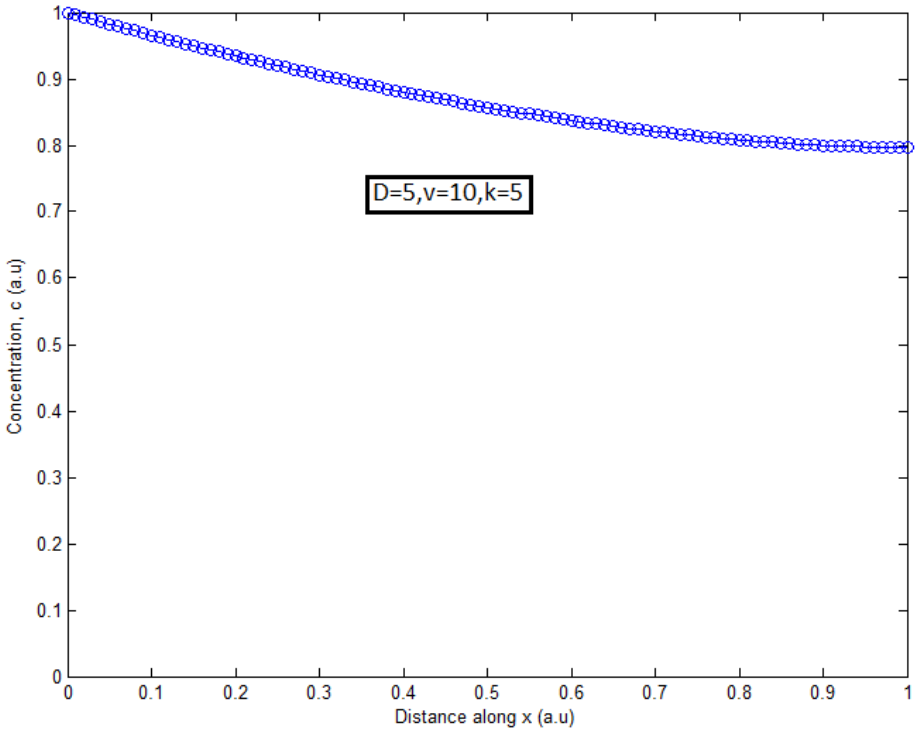

Figure 4 : Plot showing variation of c with x for D=5, v=10, k=5

Changing "D" while holding v and k constant

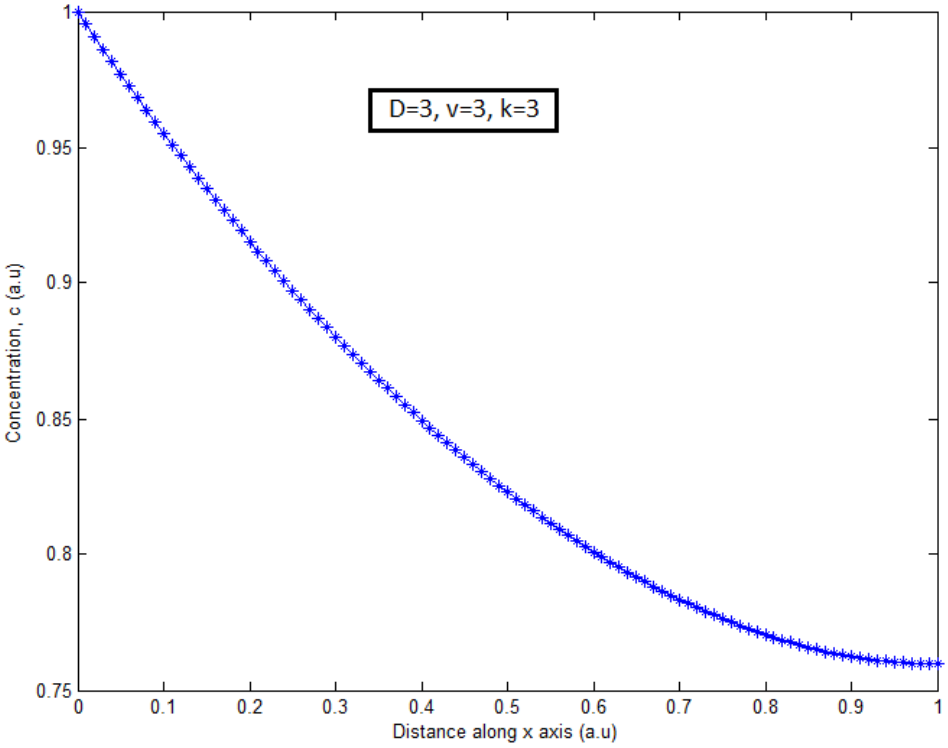

Figure 5 : Plot showing variation of c with x for D=3, v=3, k=3

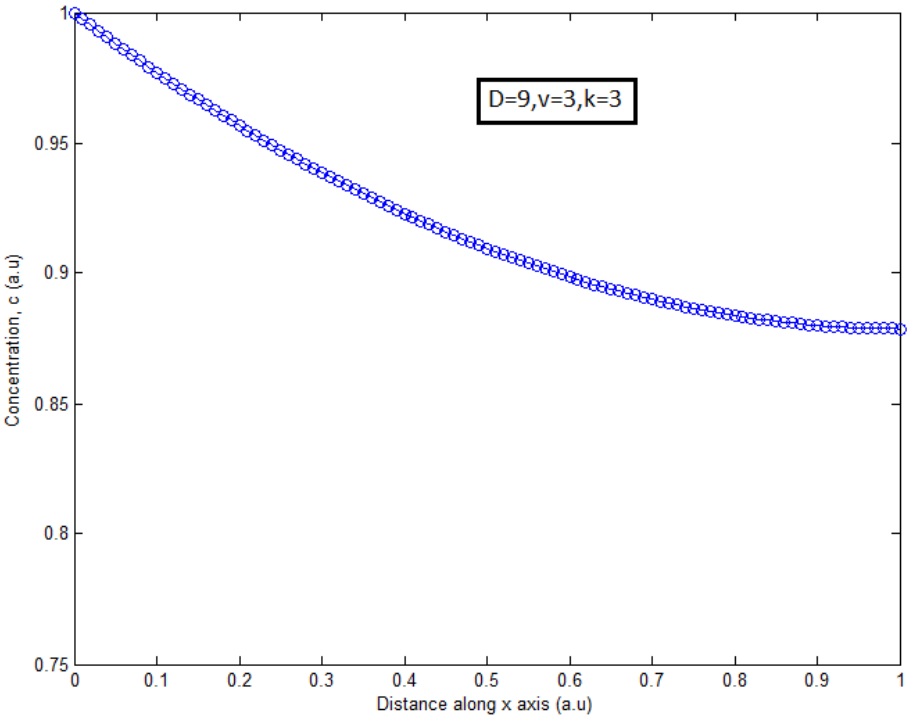

Figure 6 : Plot showing variation of c with x for D=9, v=3, k=3

Comments on the variation of parameters

For a reaction-diffusion-convection problem, the absolute values of D,k and v do not matter. It is the relative importance of convection and diffusion (the Peclet number) and the relative importance of the reaction and diffusion terms (the Damkohler number) that qualitatively affects the behavior of these graphs. All these plots were obtained by using 100 elements and setting a tolerance of 1e-4.

(c) Error Analysis

The given system was solved with 1000 elements and this solution was used as the basis with which the other solutions were compared. D,v and k were taken as 1 for the purposes of error analysis. The tolerance was set as 1e-08.

The error was computed by subtracting the value of the solution at the appropriate points and normalized by the number of elements.

The following graph is obtained between the norm of the error and h, the size of the element.

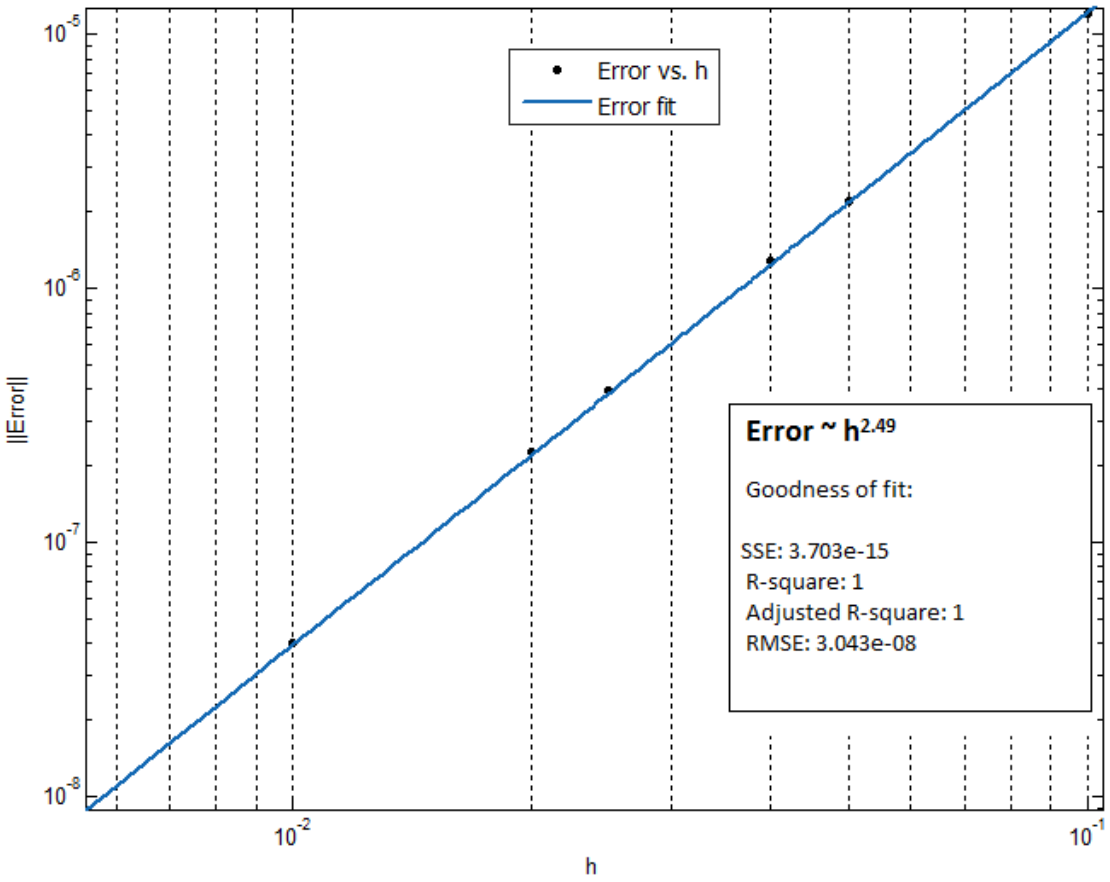

Figure : Variation of the norm of error with the size of the element

The accuracy of Finite Element method is given by an empirical formula. For linear elements, the error is expected to scale as h^2. With Finite Difference, the error is due to truncation of Taylor series and can be shown to be $O(h^2)$ accurate.

Technically, both FEM and FDM have to be equally accurate. In this case, FEM appears to be more accurate. The presence of the non-linear term might be a reason for this deviation from the expected result.

Appendix A

FEMGSolve.m

```
%*********************************************************************%
% This function solves the given one-dimensional differential equation  %
% using Finite Element Method with linear Galerkin elements.            %
% The function takes the number of elements, the Diffusivity, the       %
% velocity, the rate constant and the tolerance as input. It returns    %
% c(x) as the solution.                                                 %
%                                                                       %
%*********************************************************************%

function [sol] = FEMGsolve(N_el,D,v,k,tol)

N = N_el+1;
h =1/N_el;
syms e;

x = zeros(1,N);
for i=1:N
    x(i) = (i-1)*h;
end

c = 0.5*ones(N,1);
%Mapping between global and master element
phi(1) = 0.5*(1-e);
phi(2) = 0.5*(1+e);

dphi_de(1) = diff(phi(1),e);
dphi_de(2) = diff(phi(2),e);

norm_delc = 1;
count = 0;

c(1) = 1;  % boundary condition. Has to be true.
while(norm_delc>tol) % Beginning of Newton's loop
    J = zeros(N,N);
    R = zeros(N,1);
    J_loc = zeros(2,2);
    R_loc = zeros(2,1);
    x_loc = zeros(2,1);

    for l=1:N_el % looping over all elements
      T = sym(zeros(2,1));  % for evaluating residual
      S = sym(zeros(2,2)); % for evaluating Jacobian
      c_loc(1) = c(l);
      c_loc(2) = c(l+1);
      x_loc(1) = x(l);
      x_loc(2) = x(l+1);

      Nlin =sym(zeros(2,1));

      X = x_loc(1)*dphi_de(1) + x_loc(2)*dphi_de(2);
```

```
      for i=1:2

            for j=1:2

            T(i) = T(i) + c_loc(j)*(D*(dphi_de(i)*dphi_de(j)/X)
+v*(phi(i)*dphi_de(j)));
            if(j==1)
                flag=2;
            else flag=1;
            end
            S(i,j) = D*(dphi_de(i)*dphi_de(j)/X) +v*(phi(i)*dphi_de(j)) +
k*phi(i)*X* (2*c_loc(j)*phi(j)^2 + 2*c_loc(flag)*phi(1)*phi(2));
            J_loc(i,j) = GaussEval(S(i,j));
            end

            Nlin(i) = phi(i)*k* X*(c_loc(1)^2*phi(1)^2 +
2*c_loc(1)*c_loc(2)*phi(1)*phi(2) + c_loc(2)^2*phi(2)^2);
            R_loc(i) = GaussEval(T(i) + Nlin(i));
        end
        R(l) = R(l) + R_loc(1);
        R(l+1) = R_loc(2);

        J(l,l) = J(l,l) + J_loc(1,1);
        J(l,l+1) = J_loc(1,2);
        J(l+1,l) = J_loc(2,1);
        J(l+1,l+1) = J_loc(2,2);
    end

   R(1) = 0;
   J(1,1) = 1;
   J(1,2) = 0;
   R = -1*R;
   delc = J\R;
   c = c+delc;
   norm_delc = norm(delc);
   count=count+1;

   end
sol = c;

disp('Done with one solve');
end
```

GaussEval.m

```
%*************************************************************************%
% This function uses the 3-point Gaussian quadrature to numerically      %
% evaluate the given function between the intervals -1 and +1            %
%*************************************************************************%

function [val] = GaussEval(expression)
syms e;

gauss_weights = [(5/9.),(8/9.),(5/9.)];
gauss_points  = [-1*sqrt(3/5),0,sqrt(3/5)];
```

```
a = 0;
for i=1:3
    a = a+ (gauss_weights(i)* eval(subs(expression,e,gauss_points(i))));
end
val = a;
end
```

CalcError.m

```
%*********************************************************************%
% This function takes in 2 solutions and computes the error of the    %
% coarser solution (the one calculated using a smaller number of      %
% elements) with respect to the finer solution. The error is normalised %
% by the number of points in the coarser solution.                    %
%                                                                     %
%*********************************************************************%

function [err] = CalcError(sol_fine,sol_coarse)
l1 = length(sol_fine);
l2 = length(sol_coarse);
t= zeros(l2,1);
fact = ((l1-1)/(l2-1));
disp(fact);
for i=1:l2
    pos = 1 + (i-1)*fact;
    t(i) = ((sol_fine(pos) - sol_coarse(i)));
end
err = norm(t)/(l2);
end
```

Automate.m

```
%*********************************************************************%
% This function calls the FEMGSolve and CalcError functions           %
% repeatedly to                                                       %
% estimate the error for different solutions calculated with          %
% different number of elements. Takes in D,v and k as parameters      %
% and returns the normalised error as the answer.                     %
%                                                                     %
%*********************************************************************%

function [sol_diff] = Automate(D,v,k)

tol = 1e-8;
%calculating the finer grid of solutions
[sol_fine] = FEMGsolve(1000,D,k,v,tol);
a = FEMGsolve(10,D,k,v,tol);
b = FEMGsolve(20,D,k,v,tol);
c = FEMGsolve(25,D,k,v,tol);
d = FEMGsolve(40,D,k,v,tol);
e = FEMGsolve(50,D,k,v,tol);
f = FEMGsolve(100,D,k,v,tol);

sol_diff(1) = CalcError(sol_fine,a);
sol_diff(2) = CalcError(sol_fine,b);
sol_diff(3) = CalcError(sol_fine,c);
```

```
sol_diff(4) = CalcError(sol_fine,d);
sol_diff(5) = CalcError(sol_fine,e);
sol_diff(6) = CalcError(sol_fine,f);

end
```