The Fitzhugh-Nagumo System: F.D. and F.E. (by LehrFEM Library) Implementations

Frederico Santos Teixeira

November 25, 2010

This report presents the results of the application of the Finite Element Method in a linearized system of equations, derived from FitzHugh-Nagumo Model:

$$\partial_t u = d_u^2 \Delta u + \lambda u - \sigma v + \kappa$$
$$\tau \partial_t v = d_v^2 \Delta v + u - v$$

where the domain is $\Omega = [-1, 1]^2$ and the coefficients are $d_u^2 = 0.00028$, $lambda = 1$, $sigma = 1$, $kappa = -0.05$, $tau = 0.1$ and $d_v^2 = 0.005$.

We are considering no-flux boundary condition, i.e. $\frac{\partial u}{\partial n} = 0$ and $\frac{\partial v}{\partial n} = 0$, along with noisy initial condition around zero (but greater than zero) for both variables.

After the variational formulation, we set $u(x,t) = \sum_{i=1}^{N} \mu(t).b_N^i(x)$ and $v(x,t) = \sum_{i=1}^{N} \nu(t).b_N^i(x)$ plus a piecewise linear basis function over a triangular mesh to achieve:

$$\overrightarrow{\mu}_t = M^{-1} A \overrightarrow{\mu} + F(\overrightarrow{\mu}, \overrightarrow{\nu})$$
$$\tau \overrightarrow{\nu}_t = M^{-1} A \overrightarrow{\nu} + G(\overrightarrow{\mu}, \overrightarrow{\nu})$$

where $F(\overrightarrow{\mu}, \overrightarrow{\nu}) = \lambda \overrightarrow{\mu} - \sigma \overrightarrow{\nu} + \kappa$ and $G(\overrightarrow{\mu}, \overrightarrow{\nu}) = \overrightarrow{\mu} - \overrightarrow{\nu}$.

The matrix M and A are the Mass and Galerkin matrices, respectively, corresponding to the basis choice.

As from now we split the procedure into two different branches:

1. we create functions to compute the Mass and Galerkin matrices and we show the results using MATLAB functions;

2. we solve the whole steps using LehrFEM Library: mesh generation and refinement, assembly local elements and plot solution.

The system can be solved by means of three solvers: "ODE45" or "ODE23S" or "SemiImplicitEuler". A flag in the code leads to the user choice.

Both codes follow, along with some solutions' pictures achieved by the three different solvers. The picture on the right represents the u variable, while the picture on the left represents the v. Both of them were taken after 1000 time steps.

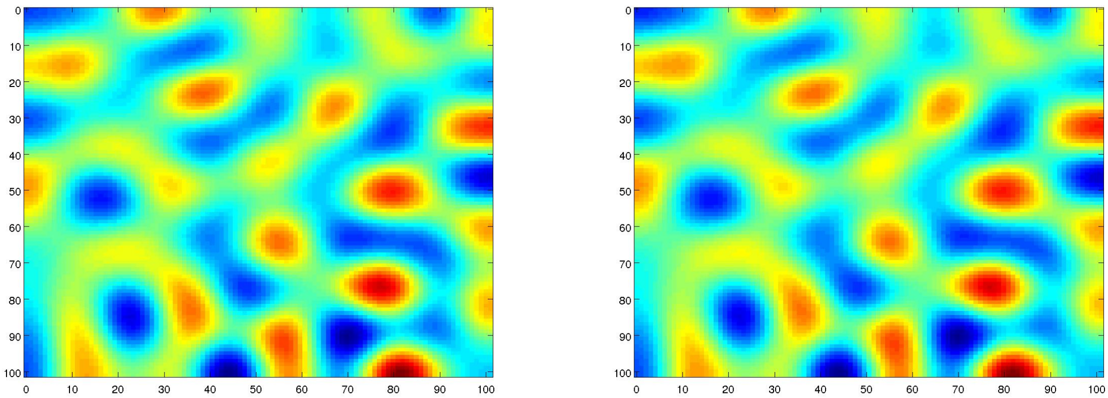
Here we present the code for the first branch:

```matlab
% ====================================== %
% Script to solve FitzHugh-Nagumo system %
%                                        %
%  u_t = d_u * Laplac(u) + F(u,v)        %
%  tau * v_t = d_v * Laplac(v) + G(u,v)  %
%                                        %
% with Neumann boundary conditions       %
% ====================================== %

close all
clear all
clc

global lambda tau du dv k sigma n N tstep tend MA time

% ---------------------
% Equation coefficients
% ---------------------

du = 0.00028;
dv = 0.005;
tau = 0.1;

% -----------------
% Functions F and G
% -----------------

lambda = 1;
k = -0.05;
sigma = 1;

F = @(u,v) lambda*u + k - sigma*v;
G = @(u,v) tau^(-1) * (u - v);

% ------------------------------------
% and its Jacobian: J = [Fu Fv; Gu Gv]
% ------------------------------------

Fu = @(u,v) lambda * ones(size(u,1), 1);
Fv = @(u,v) -sigma * ones(size(u,1), 1);

Gu = @(u,v) tau^(-1) * ones(size(u,1), 1);
Gv = @(u,v) -tau^(-1) * ones(size(u,1), 1);

% ---------------------
% Simulation parameters
% ---------------------

% number of cells in each direction.
n = 200;

% mesh discretization.
h = 1/n;

% number of unknowns.
N = (n+1)^2;

% timestep.
tstep = 0.1;

% time simulation.
tend = 100;
```

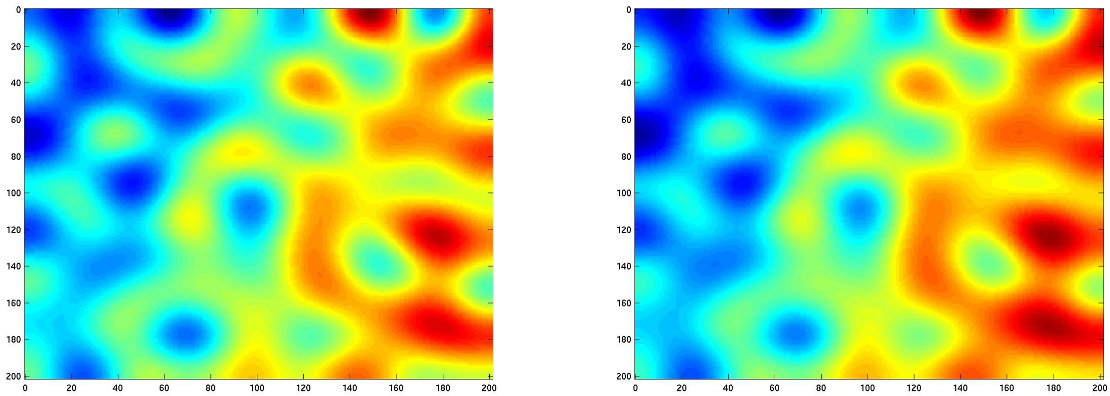
```
time = round(tend / tstep);

% ------------------------
% Initial Conditions
% ------------------------

u0 = 0.01 * rand(n+1);
v0 = 0.001 * rand(n+1);

% -----------------------------
% Mass (Diagonal) Matrix
% -----------------------------

M = MassMatrix(N, h);

% ------------------------------
% Assemble Galerkin Matrix
% ------------------------------

A = GalerkinMatrix(N);

MA = spdiags(1 ./ M, 0, N, N) * A;

% ------------------------------------------------
% Routine to solve the P.D.E. system above
% Select a method setting the 'flag' variable below
% ------------------------------------------------

flag = 'ode45';
%flag = 'ode23s';
%flag = 'ImplicitEuler';

whos
tic
v = nlevolution(u0, v0, F, G, Fu, Fv, Gu, Gv, flag);
toc

%----------------
% Solution plot
%----------------

fig = figure();

for i = 1:size(v, 1)
    mi_end = reshape(v(i, 1:N), (n+1), (n+1));
    nu_end = reshape(v(i, N+1:end), (n+1), (n+1));

    imagesc(0:(n+1), 0:(n+1), mi_end);
    print(fig, '-djpeg', sprintf('mi_end%5d', i) );

    imagesc(0:(n+1), 0:(n+1), nu_end);
    print(fig, '-djpeg', sprintf('nu_end%5d', i) );
end
```

The results follow below:

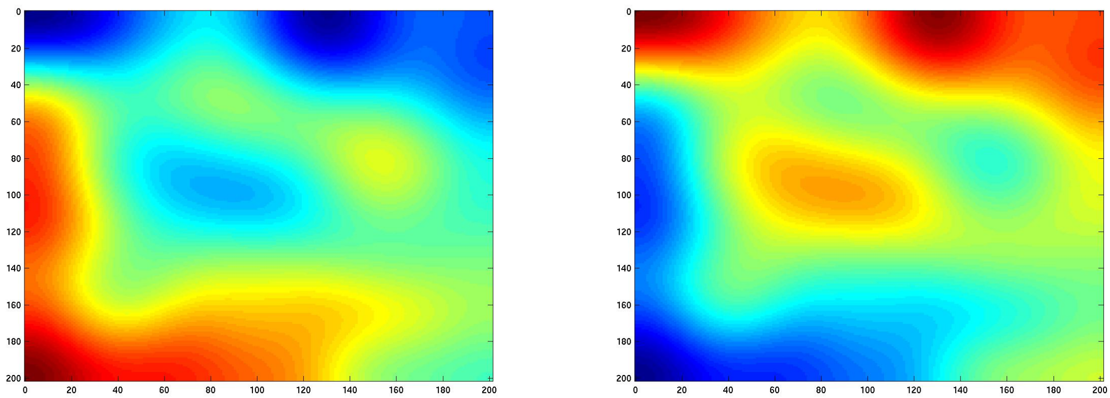
u and v by means of "ODE45":



u and v by means of "ODE23S":



u and v by means of "ImplicitEuler":



There are two functions which generate the Mass and Galerkin Matrices:

```
function [ M ] = MassMatrix( N, h )

```

```
% ---------------------------
% Mass (Diagonal) Matrix
% ---------------------------

M = h^2 * ones(N, 1);

n = round(sqrt(N) - 1);

M(1,:) = M(1,:)/3;
M(N,:) = M(N,:)/3;

M(2:n,:) = M(2:n,:)/2;
M((n*(n+1)+2):(n*(n+1)+n),:) = M((n*(n+1)+2):(n*(n+1)+n),:)/2;

M(n+1,:) = M(n+1,:)/6;
M(n*(n+1)+1,:) = M(n*(n+1)+1,:)/6;

for k=2:n
    M(k*(n+1),:) = M(k*(n+1),:)/2;
    M((k-1)*(n+1)+1,:) = M((k-1)*(n+1)+1,:)/2;
end

end
```

```
function [ A ] = GalerkinMatrix( N )

% ----------------------------
% Assemble Galerkin Matrix
% ----------------------------

A = spalloc(N, N, 8*N);

n = round(sqrt(N)-1);

for i = 1:(n+1):(n^2)
    for j = 0:(n-1)
        m = i+j;

        A([m m+1 m+1+n+1], [m m+1 m+1+n+1]) = ...
            A([m m+1 m+1+n+1], [m m+1 m+1+n+1]) + ...
            [ -0.5   0.5    0;...
               0.5  -1      0.5; ...
               0     0.5   -0.5 ];

        A([m m+n+1 m+1+n+1], [m m+n+1 m+1+n+1]) = ...
            A([m m+n+1 m+1+n+1], [m m+n+1 m+1+n+1]) + ...
            [ -0.5   0.5    0;...
               0.5  -1      0.5; ...
               0     0.5   -0.5 ];
    end;
end

end
```

Now the code for the second branch:

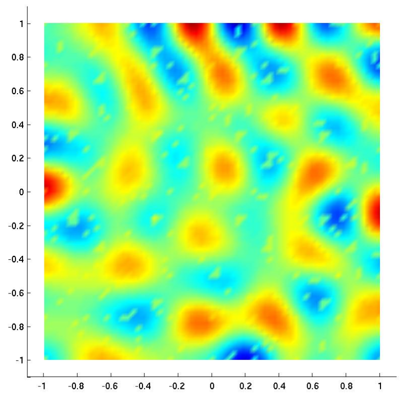
```
% ====================================== %
% Script to solve FitzHugh-Nagumo system %
%                                        %
%  u_t = d_u * Laplac(u) + F(u,v)        %
```

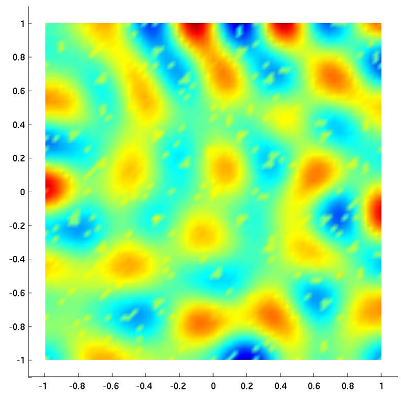
```
%  tau * v_t = d_v * Laplac(v) + G(u,v)   %
%                                         %
% with Neumann boundary conditions.       %
% by means of LehrFEM                     %
% ======================================= %

close all
clear all
clc

global lambda tau du dv k sigma n N tstep tend MA time

% ---------------------
% Equation coefficients
% ---------------------

du = 0.00028;
dv = 0.005;
tau = 0.1;

% -----------------
% Functions F and G
% -----------------

lambda = 1;
k = -0.05;
sigma = 1;

F = @(u,v) lambda*u + k - sigma*v;
G = @(u,v) tau^(-1) * (u - v);

% ------------------------------------
% and its Jacobian: J = [Fu Fv; Gu Gv]
% ------------------------------------

Fu = @(u,v) lambda * ones(size(u,1), 1);
Fv = @(u,v) -sigma * ones(size(u,1), 1);

Gu = @(u,v) tau^(-1) * ones(size(u,1), 1);
Gv = @(u,v) -tau^(-1) * ones(size(u,1), 1);

% ---------------------
% Simulation parameters
% ---------------------

% number of cells in each direction.
n1 = 50;

% timestep.
tstep = 0.1;

% time simulation.
tend = 1000;

time = round(tend/tstep);

% ---------------
% Initialize mesh
% ---------------

Mesh = load_Mesh('Coord_Sqr.dat','Elem_Sqr.dat');
Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
```

```
Mesh = add_Edges(Mesh);
Loc = get_BdEdges(Mesh);
Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
Mesh.BdFlags(Loc) = -1;
Mesh = add_Edge2Elem(Mesh);

%m = size(Mesh.Elements, 1);
%RefNum = 4;

while m < 1:(n1^2)
    Mesh = refine_REG(Mesh);
    m = size(Mesh.Elements,1);
end

Mesh = add_Edge2Elem(Mesh);
%plot_Mesh(Mesh);

% number of unknowns:
N = size(Mesh.Coordinates, 1);

% new number of cells in each direction.
n = round(sqrt(N) - 1);

% mesh discretization.
h = 1/n;

% ------------------
% Initial Conditions
% ------------------

u0 = 0.01 * rand(n+1);
v0 = 0.001 * rand(n+1);

%[u0, v0] = InitialCondition(N);

% ----------------------
% Mass (Diagonal) Matrix
% ----------------------

M = MassMatrix(N, h);

% -------------------------
% Assemble Galerkin Matrix
% -------------------------

A = assemMat_LFE(Mesh, @STIMA_LaplFHN_LFE);

MA = A * spdiags(1 ./ M, 0, N, N);

% ---------------------------------------------------
% Routine to solve the P.D.E. system above
% Select a method setting the 'flag' variable above
% ---------------------------------------------------

%flag = 'ode45';
%flag = 'ode23s';
flag = 'ImplicitEuler';

whos

tic
v = nlevolution(u0, v0, F, G, Fu, Fv, Gu, Gv, flag);
```

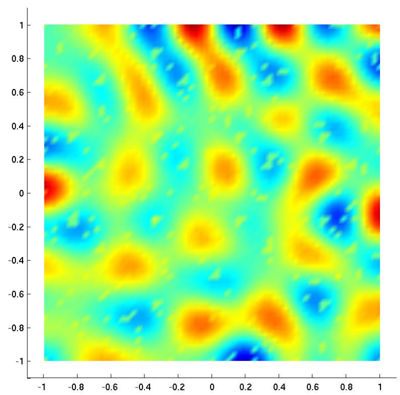
```
toc

%----------------
% Solution plot
%----------------

fig = figure('visible', 'off');

%for i = 1:size(v, 1)
    H = plot_LFE(v(end, 1:N)', Mesh, fig);
    print(H, '-djpeg', sprintf('mi_LehrFEM_%delements_step0.01_time50', N) );

    J = plot_LFE(v(end, N+1:end)', Mesh, fig);
    print(J, '-djpeg', sprintf('nu_LehrFEM_%delements_step0.01_time50', N) );
%end
```

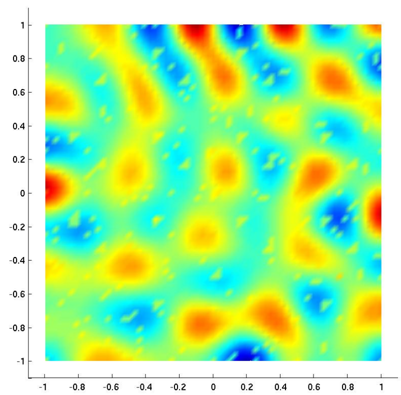
The results follow below:
u and v by means of "ODE45":



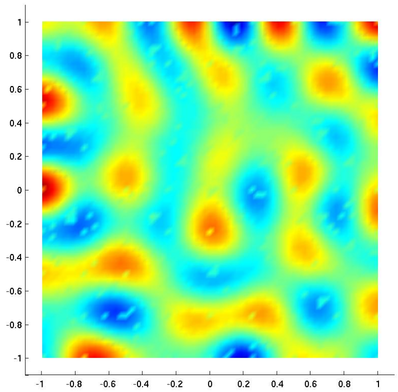


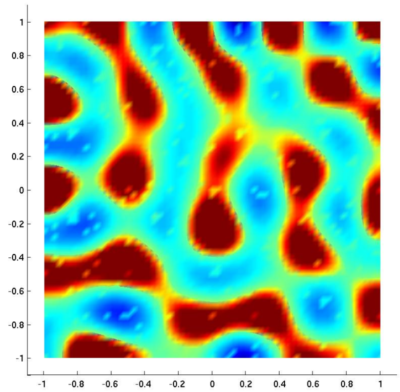
u and v by means of "ODE23S":





u and v by means of "ImplicitEuler":





The code showed below is used to solve this system:

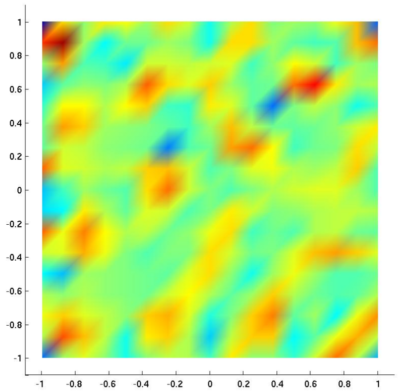
```
function [ v ] = nlevolution( u0, v0, FHandle, GHandle, FuHandle, ...
                              FvHandle, GuHandle, GvHandle, flag )

% ------------------------------------------------------------------
% INPUTS:
% ------
%
% 'u0' and 'v0' are the initial conditions
%
% 'FHandle' and 'GHandle' are the functions handles
%
% J = [FuHandle FvHandle; GuHandle GvHandle] is the Jacobian
%
% 'TEND' is the time simulation.
%
% 'flag' switchs between the possible solvers
%
% OUTPUT:
% ------
%
% v: Matrix which each line vector is a solution in a timestep.
% ------------------------------------------------------------------

global N tstep tend du dv tau MA lambda k sigma time

% ----------------------------------------------------
% Handle to function 'f' such that y'(t) = f(t, y)
% ----------------------------------------------------

f = @(t, y) [du * MA * y(1:N) + FHandle(y(1:N), y(N+1:end)) ; ...
             dv * tau^(-1) * MA * y(N+1:end) + ...
             tau^(-1) * GHandle(y(1:N), y(N+1:end))];

% -----------------------------
% Assemble initial conditions
% -----------------------------

mi0 = reshape(u0, 1, N);
nu0 = reshape(v0, 1, N);

v = [mi0 nu0];
```

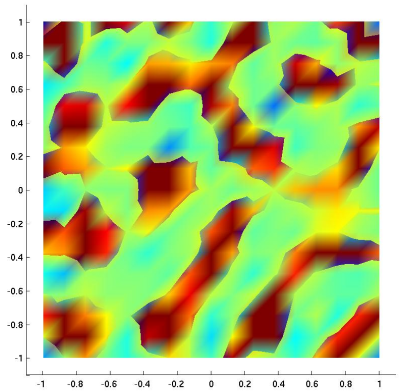
```
% ------------------------------
% Solution by means of ODE45
% ------------------------------

if (strcmp(flag, 'ode45') )

    [¬, v] = ode45(f, 0:tstep:tend, v);

end

% -------------------------------
% Solution by means of ODE23S
% -------------------------------

if ( strcmp(flag,'ode23s') )

    Jacobian=@(t, y)[du*MA+spdiags(FuHandle(y(1:N),y(N+1:end)),0,N,N), ...
             spdiags(FvHandle(y(1:N), y(N+1:end)), 0, N, N); ...
             spdiags(GuHandle(y(1:N), y(N+1:end)), 0, N, N), ...
             dv*tau^(-1)*MA+spdiags(GvHandle(y(1:N),y(N+1:end)),0,N,N)];

    options = odeset('AbsTol', 1e-5, 'RelTol', 1e-3, ...
        'Jacobian', @(t,y)Jacobian(t,y), 'JPattern', @(S)jpattern(v));

    [¬, v] = ode23s(f, 0:tstep:tend, v, options);

end

% -------------------------------------------
% Solution by means of SemiImplicitEuler
% -------------------------------------------

if( strcmp(flag,'ImplicitEuler') )

    mi_pre = mi0';
    nu_pre = nu0';

    mi1 = spdiags( (1-tstep*lambda)*ones(N,1), 0, N, N ) - tstep*du*MA;
    nu1 = spdiags( tstep*sigma*ones(N,1), 0, N,N );

    mi2 = spdiags( -tstep*ones(N,1), 0, N, N );
    nu2 = spdiags( (tau+tstep) * ones(N,1), 0, N, N ) - tstep*dv*MA;

    H = sparse([mi1, nu1; mi2, nu2 ]);

    for i = 2:time
        tic

        Y = H \ [mi_pre + tstep*k; tau * nu_pre];

        v = [v; ...
            Y(1:N)' Y(N+1:end)'];

        mi_pre = Y(1:N);
        nu_pre = Y(N+1:end);

        toc

        display( sprintf('step %d finished...', i) )
    end
end
```

```
end

function [ S ] = jpattern( v )
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

pattern = DFN(1,v');
[a, b] = size(pattern);
S = spalloc( a, b, a*b);

for I = 1:size(pattern)
    for J = 1:size(pattern)
        if (pattern(I,J) ≠ 0)
           S(I,J) = 1;
        end
    end
end

end
```

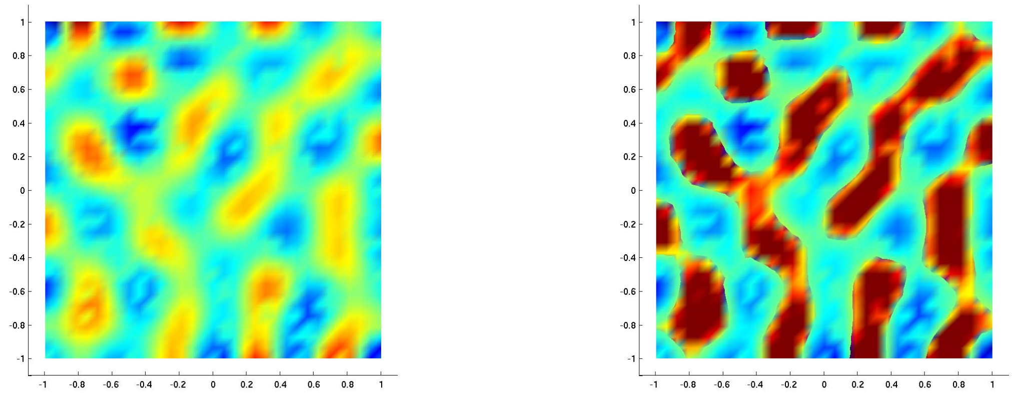
Also, fixing a initial condition and giving a time step, we can evaluate the behavior of the solution after a mesh refinement.

1. u and v for $timestep = 0.1$, $time = 50$ and 289 elements in the mesh:

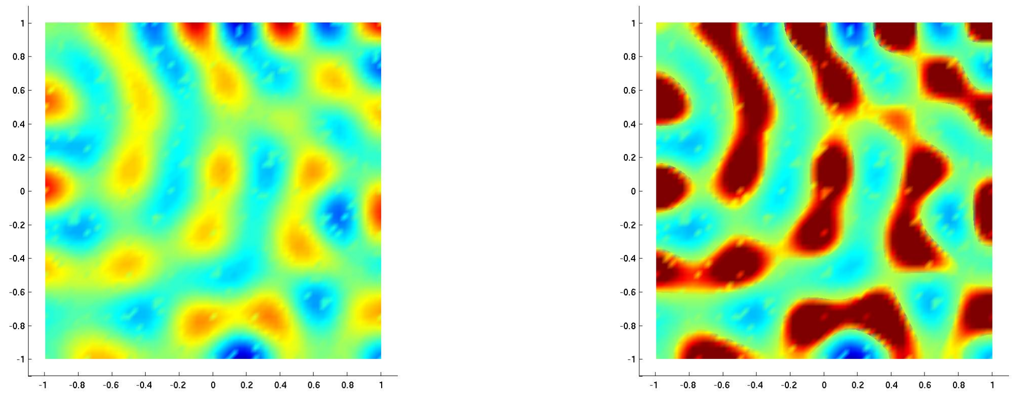




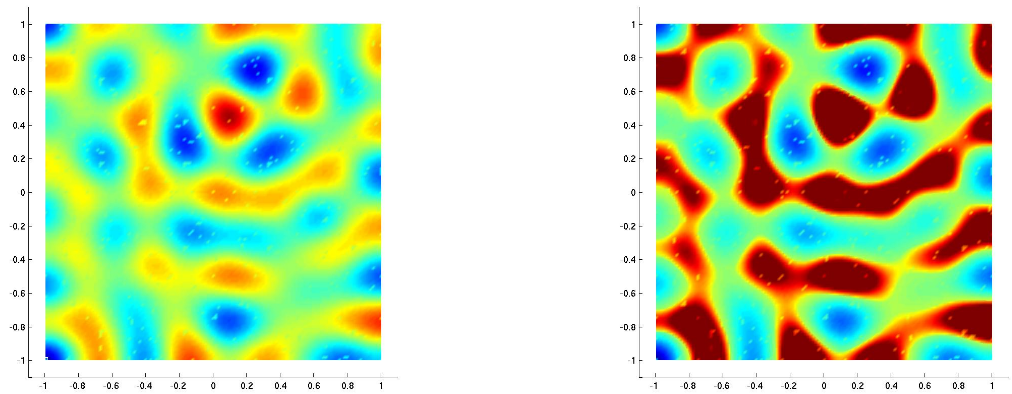
2. u and v for $timestep = 0.1$, $time = 50$ and 1089 elements in the mesh:



3. u and v for $timestep = 0.1$, $time = 50$ and 4225 elements in the mesh:

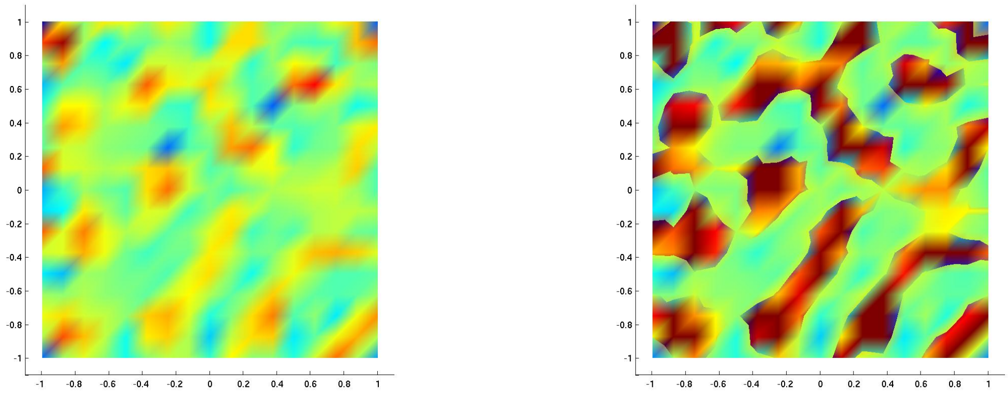


4. u and v for $timestep = 0.1$, $time = 50$ and 16641 elements in the mesh:

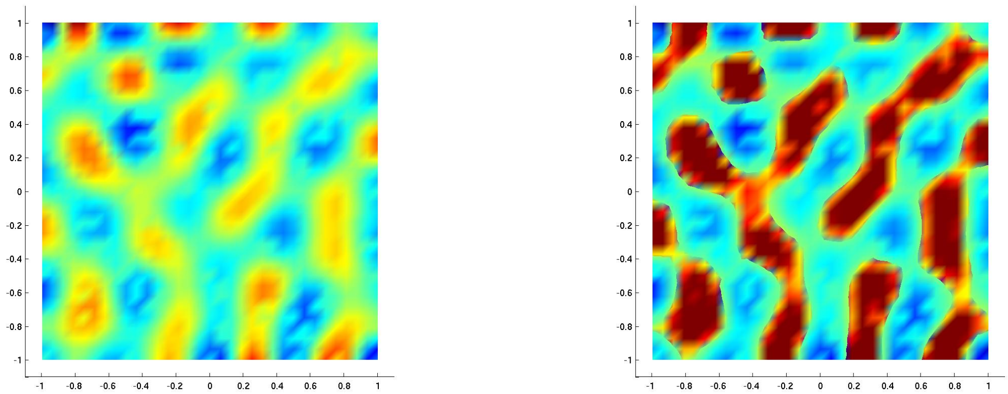


For the next tests we set a new time step.

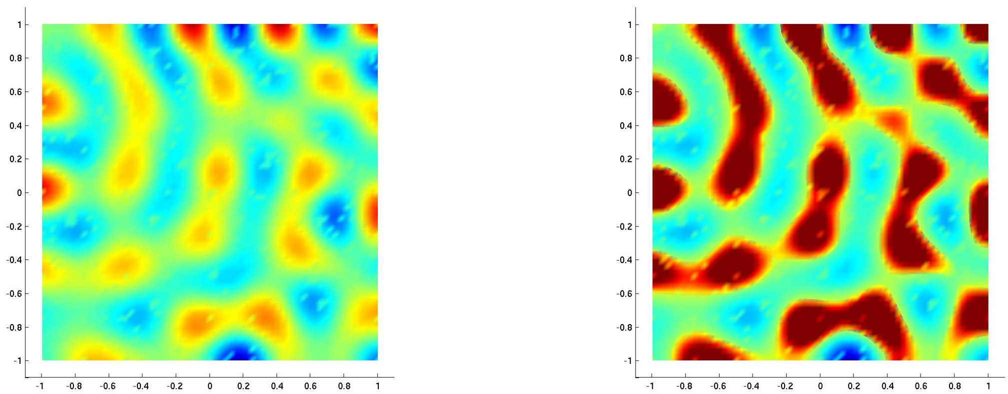
1. u and v for $timestep = 0.01$, $time = 50$ and 289 elements in the mesh:



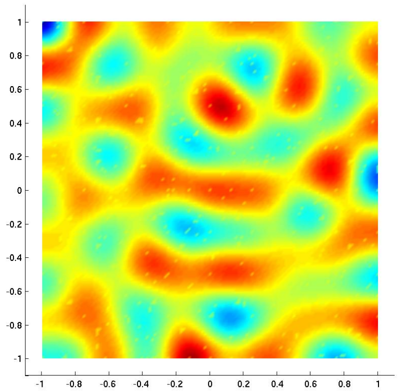
2. u and v for $timestep = 0.01$, $time = 50$ and 1089 elements in the mesh:

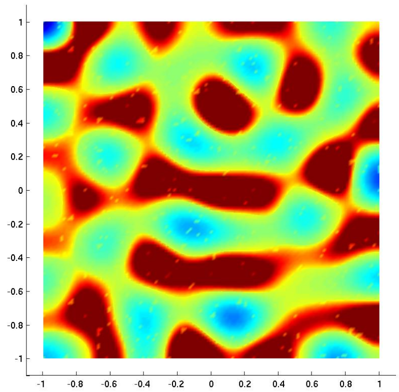


3. u and v for $timestep = 0.01$, $time = 50$ and 4225 elements in the mesh:



4. u and v for $timestep = 0.01$, $time = 50$ and 16641 elements in the mesh:





We fixed the initial condition given by the function:

```
function [ u0, v0 ] = InitialCondition( N )

u0 = zeros( round(sqrt(N)) );
v0 = zeros( round(sqrt(N)) );

u0(:,:) = 0.005;
v0(:,:) = 0.0005;

for i = 1:2:( round(sqrt(N)) )
    u0(i, :) = 0.01;
    v0(i, :) = 0.001;
end

end
```

This sort of results can be achieved for each solver described above.