

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:

$$\begin{aligned}\partial_t u &= d_u^2 \Delta u + \lambda u - \sigma v + \kappa \\ \tau \partial_t v &= d_v^2 \Delta v + u - v\end{aligned}$$

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

$$\begin{aligned}\vec{\mu}_t &= M^{-1} A \vec{\mu} + F(\vec{\mu}, \vec{\nu}) \\ \tau \vec{\nu}_t &= M^{-1} A \vec{\nu} + G(\vec{\mu}, \vec{\nu})\end{aligned}$$

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

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:

```

1  % ===== %
2  % Script to solve FitzHugh-Nagumo system %
3  % %
4  %  $u_t = d_u * \text{Laplac}(u) + F(u,v)$  %
5  %  $\tau * v_t = d_v * \text{Laplac}(v) + G(u,v)$  %
6  % %
7  % with Neumann boundary conditions %
8  % ===== %
9
10 close all
11 clear all
12 clc
13
14 global lambda tau du dv k sigma n N tstep tend MA time
15
16 % -----
17 % Equation coefficients
18 % -----
19
20 du = 0.00028;
21 dv = 0.005;
22 tau = 0.1;
23
24 % -----
25 % Functions F and G
26 % -----
27
28 lambda = 1;
29 k = -0.05;
30 sigma = 1;
31
32 F = @(u,v) lambda*u + k - sigma*v;
33 G = @(u,v) tau^(-1) * (u - v);
34
35 % -----
36 % and its Jacobian: J = [Fu Fv; Gu Gv]
37 % -----
38
39 Fu = @(u,v) lambda * ones(size(u,1), 1);
40 Fv = @(u,v) -sigma * ones(size(u,1), 1);
41
42 Gu = @(u,v) tau^(-1) * ones(size(u,1), 1);
43 Gv = @(u,v) -tau^(-1) * ones(size(u,1), 1);
44
45 % -----
46 % Simulation parameters
47 % -----
48
49 % number of cells in each direction.
50 n = 200;
51
52 % mesh discretization.
53 h = 1/n;
54
55 % number of unknowns.
56 N = (n+1)^2;
57
58 % timestep.
59 tstep = 0.1;
60
61 % time simulation.
62 tend = 100;

```

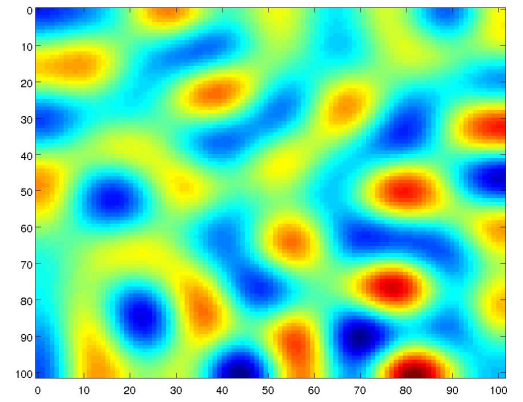
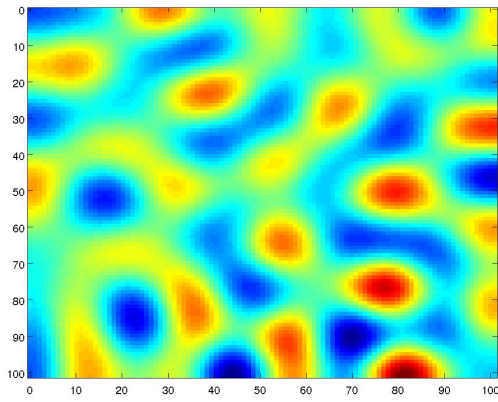
```

63
64 time = round(tend / tstep);
65
66 % -----
67 % Initial Conditions
68 % -----
69
70 u0 = 0.01 * rand(n+1);
71 v0 = 0.001 * rand(n+1);
72
73 % -----
74 % Mass (Diagonal) Matrix
75 % -----
76
77 M = MassMatrix(N, h);
78
79 % -----
80 % Assemble Galerkin Matrix
81 % -----
82
83 A = GalerkinMatrix(N);
84
85 MA = spdiags(1 ./ M, 0, N, N) * A;
86
87 % -----
88 % Routine to solve the P.D.E. system above
89 % Select a method setting the 'flag' variable below
90 % -----
91
92 flag = 'ode45';
93 %flag = 'ode23s';
94 %flag = 'ImplicitEuler';
95
96 whos
97 tic
98 v = nlevolution(u0, v0, F, G, Fu, Fv, Gu, Gv, flag);
99 toc
100
101 % -----
102 % Solution plot
103 % -----
104
105 fig = figure();
106
107 for i = 1:size(v, 1)
108     mi_end = reshape(v(i, 1:N), (n+1), (n+1));
109     nu_end = reshape(v(i, N+1:end), (n+1), (n+1));
110
111     imagesc(0:(n+1), 0:(n+1), mi_end);
112     print(fig, '-djpeg', sprintf('mi_end%5d', i) );
113
114     imagesc(0:(n+1), 0:(n+1), nu_end);
115     print(fig, '-djpeg', sprintf('nu_end%5d', i) );
116 end

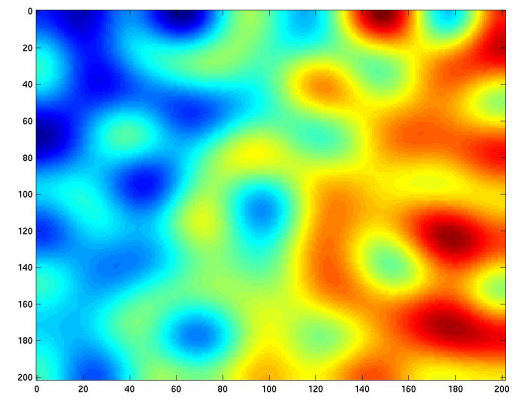
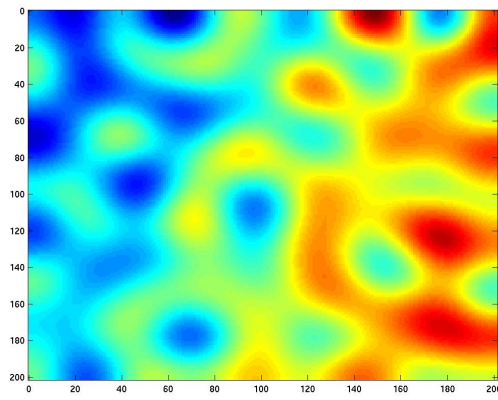
```

The results follow below:

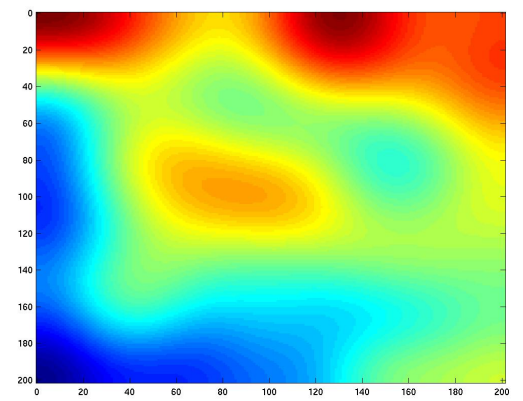
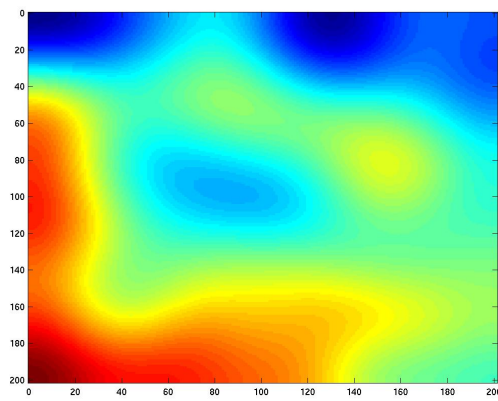
u and v by means of "ODE45":



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:

```
1 function [ M ] = MassMatrix( N, h )
2
```

```

3  % -----
4  % Mass (Diagonal) Matrix
5  % -----
6
7  M = h^2 * ones(N, 1);
8
9  n = round(sqrt(N) - 1);
10
11 M(1,:) = M(1,+)/3;
12 M(N,:) = M(N,+)/3;
13
14 M(2:n,:) = M(2:n,+)/2;
15 M((n*(n+1)+2):(n*(n+1)+n), :) = M((n*(n+1)+2):(n*(n+1)+n),+)/2;
16
17 M(n+1,:) = M(n+1,+)/6;
18 M(n*(n+1)+1,:) = M(n*(n+1)+1,+)/6;
19
20 for k=2:n
21     M(k*(n+1), :) = M(k*(n+1),+)/2;
22     M((k-1)*(n+1)+1, :) = M((k-1)*(n+1)+1,+)/2;
23 end
24
25 end

```

```

1  function [ A ] = GalerkinMatrix( N )
2
3  % -----
4  % Assemble Galerkin Matrix
5  % -----
6
7  A = spalloc(N, N, 8*N);
8
9  n = round(sqrt(N)-1);
10
11 for i = 1:(n+1):(n^2)
12     for j = 0:(n-1)
13         m = i+j;
14
15         A([m m+1 m+1+n+1], [m m+1 m+1+n+1]) = ...
16             A([m m+1 m+1+n+1], [m m+1 m+1+n+1]) + ...
17             [ -0.5    0.5    0;...
18               0.5   -1    0.5; ...
19               0    0.5   -0.5 ];
20
21         A([m m+n+1 m+1+n+1], [m m+n+1 m+1+n+1]) = ...
22             A([m m+n+1 m+1+n+1], [m m+n+1 m+1+n+1]) + ...
23             [ -0.5    0.5    0;...
24               0.5   -1    0.5; ...
25               0    0.5   -0.5 ];
26     end;
27 end
28
29 end

```

Now the code for the second branch:

```

1  % ===== %
2  % Script to solve FitzHugh-Nagumo system %
3  % %
4  % u_t = d_u * Laplac(u) + F(u,v) %

```

```

5 % tau * v_t = d_v * Laplac(v) + G(u,v) %
6 % %
7 % with Neumann boundary conditions. %
8 % by means of LehrFEM %
9 % ===== %
10
11 close all
12 clear all
13 clc
14
15 global lambda tau du dv k sigma n N tstep tend MA time
16
17 % -----
18 % Equation coefficients
19 % -----
20
21 du = 0.00028;
22 dv = 0.005;
23 tau = 0.1;
24
25 % -----
26 % Functions F and G
27 % -----
28
29 lambda = 1;
30 k = -0.05;
31 sigma = 1;
32
33 F = @(u,v) lambda*u + k - sigma*v;
34 G = @(u,v) tau^(-1) * (u - v);
35
36 % -----
37 % and its Jacobian: J = [Fu Fv; Gu Gv]
38 % -----
39
40 Fu = @(u,v) lambda * ones(size(u,1), 1);
41 Fv = @(u,v) -sigma * ones(size(u,1), 1);
42
43 Gu = @(u,v) tau^(-1) * ones(size(u,1), 1);
44 Gv = @(u,v) -tau^(-1) * ones(size(u,1), 1);
45
46 % -----
47 % Simulation parameters
48 % -----
49
50 % number of cells in each direction.
51 nl = 50;
52
53 % timestep.
54 tstep = 0.1;
55
56 % time simulation.
57 tend = 1000;
58
59 time = round(tend/tstep);
60
61 % -----
62 % Initialize mesh
63 % -----
64
65 Mesh = load_Mesh('Coord_Sqr.dat', 'Elem_Sqr.dat');
66 Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);

```

```

67 Mesh = add_Edges(Mesh);
68 Loc = get_BdEdges(Mesh);
69 Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
70 Mesh.BdFlags(Loc) = -1;
71 Mesh = add_Edge2Elem(Mesh);
72
73 %m = size(Mesh.Elements, 1);
74 %RefNum = 4;
75
76 while m < 1:(n1^2)
77     Mesh = refine_REG(Mesh);
78     m = size(Mesh.Elements,1);
79 end
80
81 Mesh = add_Edge2Elem(Mesh);
82 %plot_Mesh(Mesh);
83
84 % number of unknowns:
85 N = size(Mesh.Coordinates, 1);
86
87 % new number of cells in each direction.
88 n = round(sqrt(N) - 1);
89
90 % mesh discretization.
91 h = 1/n;
92
93 % -----
94 % Initial Conditions
95 % -----
96
97 u0 = 0.01 * rand(n+1);
98 v0 = 0.001 * rand(n+1);
99
100 % [u0, v0] = InitialCondition(N);
101
102 % -----
103 % Mass (Diagonal) Matrix
104 % -----
105
106 M = MassMatrix(N, h);
107
108 % -----
109 % Assemble Galerkin Matrix
110 % -----
111
112 A = assemMat_LFE(Mesh, @STIMA_LaplFHN_LFE);
113
114 MA = A * spdiags(1 ./ M, 0, N, N);
115
116 % -----
117 % Routine to solve the P.D.E. system above
118 % Select a method setting the 'flag' variable above
119 % -----
120
121 %flag = 'ode45';
122 %flag = 'ode23s';
123 flag = 'ImplicitEuler';
124
125 whos
126
127 tic
128 v = nlevolution(u0, v0, F, G, Fu, Fv, Gu, Gv, flag);

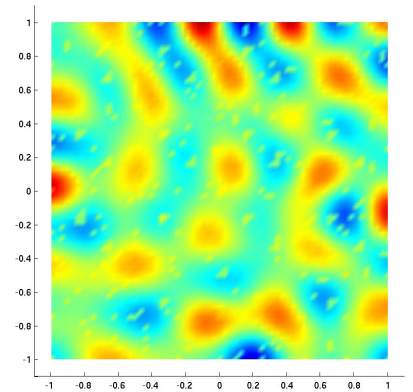
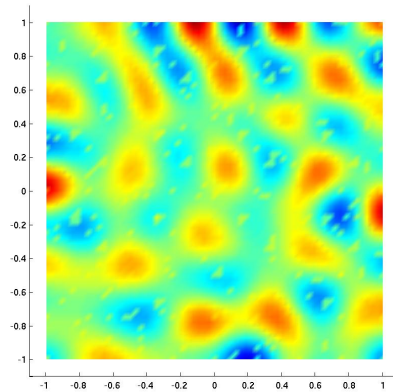
```

```

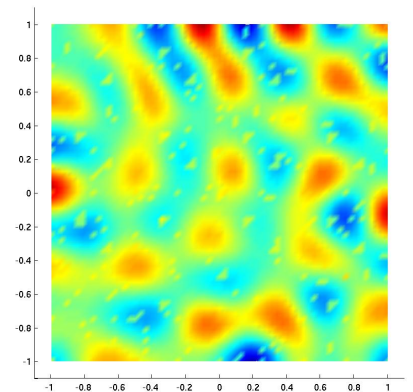
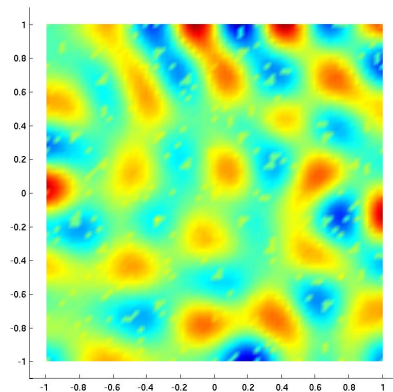
129 toc
130
131 %-----
132 % Solution plot
133 %-----
134
135 fig = figure('visible', 'off');
136
137 %for i = 1:size(v, 1)
138     H = plot_LFE(v(end, 1:N)', Mesh, fig);
139     print(H, '-djpeg', sprintf('mi_LehrFEM_%delements_step0.01_time50', N) );
140
141     J = plot_LFE(v(end, N+1:end)', Mesh, fig);
142     print(J, '-djpeg', sprintf('nu_LehrFEM_%delements_step0.01_time50', N) );
143 %end

```

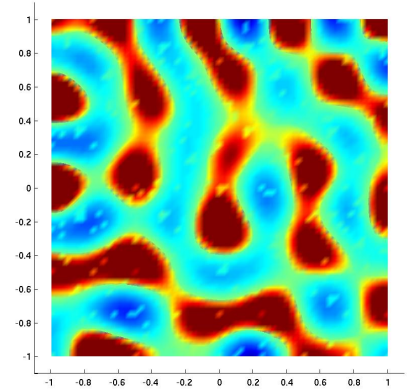
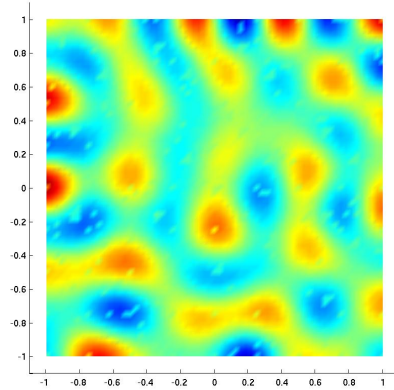
The results follow below:
 u and v by means of "ODE45":



u and v by means of "ODE23S":



u and v by means of "ImplicitEuler":



The code showed below is used to solve this system:

```

1 function [ v ] = nlevolution( u0, v0, FHandle, GHandle, FuHandle, ...
2                               FvHandle, GuHandle, GvHandle, flag )
3
4 % -----
5 % INPUTS:
6 % -----
7 %
8 % 'u0' and 'v0' are the initial conditions
9 %
10 % 'FHandle' and 'GHandle' are the functions handles
11 %
12 % J = [FuHandle FvHandle; GuHandle GvHandle] is the Jacobian
13 %
14 % 'TEND' is the time simulation.
15 %
16 % 'flag' switches between the possible solvers
17 %
18 % OUTPUT:
19 % -----
20 %
21 % v: Matrix which each line vector is a solution in a timestep.
22 % -----
23
24 global N tstep tend du dv tau MA lambda k sigma time
25
26 % -----
27 % Handle to function 'f' such that y'(t) = f(t, y)
28 % -----
29
30 f = @(t, y) [du * MA * y(1:N) + FHandle(y(1:N), y(N+1:end)) ; ...
31              dv * tau^(-1) * MA * y(N+1:end) + ...
32              tau^(-1) * GHandle(y(1:N), y(N+1:end))];
33
34 % -----
35 % Assemble initial conditions
36 % -----
37
38 mi0 = reshape(u0, 1, N);
39 nu0 = reshape(v0, 1, N);
40
41 v = [mi0 nu0];

```

```

42
43 % -----
44 % Solution by means of ODE45
45 % -----
46
47 if (strcmp(flag, 'ode45') )
48
49     [~, v] = ode45(f, 0:tstep:tend, v);
50
51 end
52
53 % -----
54 % Solution by means of ODE23S
55 % -----
56
57 if ( strcmp(flag,'ode23s') )
58
59     Jacobian=@(t, y) [du*MA+spdiags (FuHandle(y(1:N),y(N+1:end)),0,N,N), ...
60         spdiags(FvHandle(y(1:N), y(N+1:end)), 0, N, N); ...
61         spdiags(GuHandle(y(1:N), y(N+1:end)), 0, N, N), ...
62         dv*tau^(-1)*MA+spdiags (GvHandle(y(1:N),y(N+1:end)),0,N,N)];
63
64     options = odeset('AbsTol', 1e-5, 'RelTol', 1e-3, ...
65         'Jacobian', @(t,y)Jacobian(t,y), 'JPattern', @(S) jpattern(v));
66
67     [~, v] = ode23s(f, 0:tstep:tend, v, options);
68
69 end
70
71 % -----
72 % Solution by means of SemiImplicitEuler
73 % -----
74
75 if( strcmp(flag,'ImplicitEuler') )
76
77     mi_pre = mi0';
78     nu_pre = nu0';
79
80     mi1 = spdiags( (1-tstep*lambda)*ones(N,1), 0, N, N ) - tstep*du*MA;
81     nu1 = spdiags( tstep*sigma*ones(N,1), 0, N, N );
82
83     mi2 = spdiags( -tstep*ones(N,1), 0, N, N );
84     nu2 = spdiags( (tau+tstep) * ones(N,1), 0, N, N ) - tstep*dv*MA;
85
86     H = sparse([mi1, nu1; mi2, nu2 ]);
87
88     for i = 2:time
89         tic
90
91         Y = H \ [mi_pre + tstep*k; tau * nu_pre];
92
93         v = [v; ...
94             Y(1:N)' Y(N+1:end)'];
95
96         mi_pre = Y(1:N);
97         nu_pre = Y(N+1:end);
98
99         toc
100
101         display( sprintf('step %d finished...', i) )
102     end
103 end

```

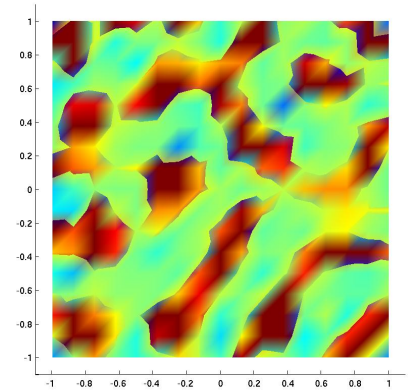
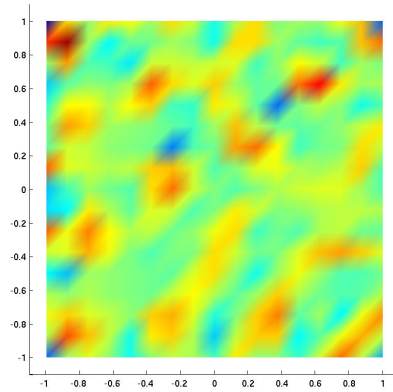
```

104
105 end
106
107 function [ S ] = jpattern( v )
108 %UNTITLED Summary of this function goes here
109 % Detailed explanation goes here
110
111 pattern = DFN(1,v');
112 [a, b] = size(pattern);
113 S = spalloc( a, b, a*b);
114
115 for I = 1:size(pattern)
116     for J = 1:size(pattern)
117         if (pattern(I,J)  $\neq$  0)
118             S(I,J) = 1;
119         end
120     end
121 end
122
123 end

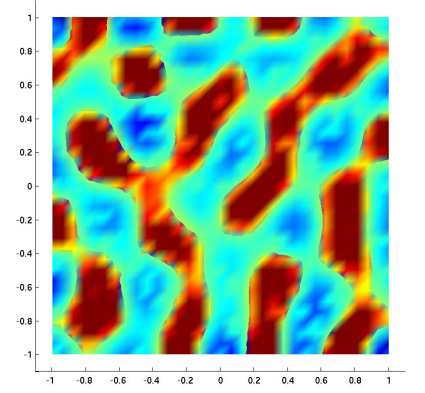
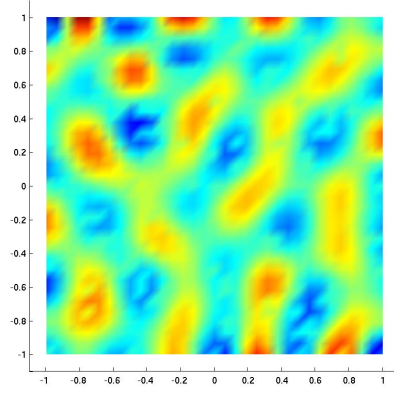
```

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

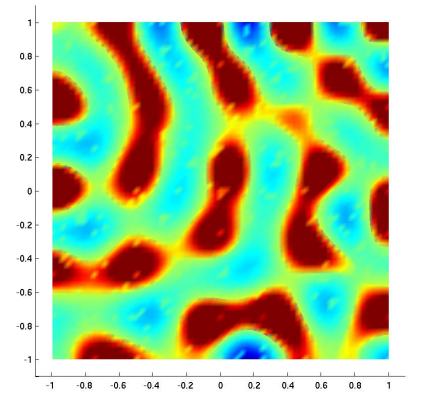
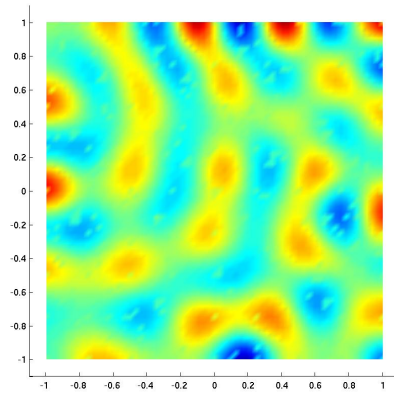
1. u and v for $timestep = 0.1$, $time = 50$ and 289 elements in the mesh:



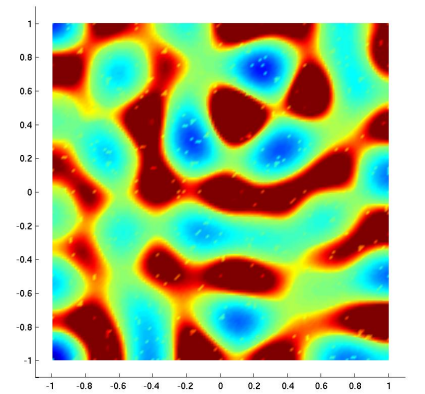
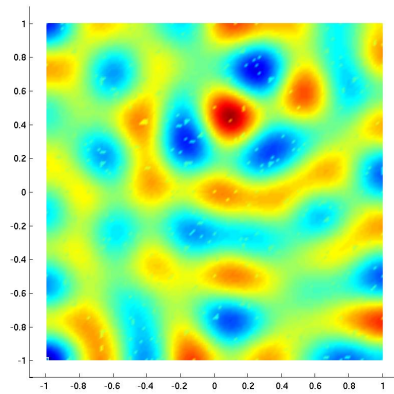
2. u and v for $timestep = 0.1$, $time = 50$ and 1089 elements in the mesh:



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

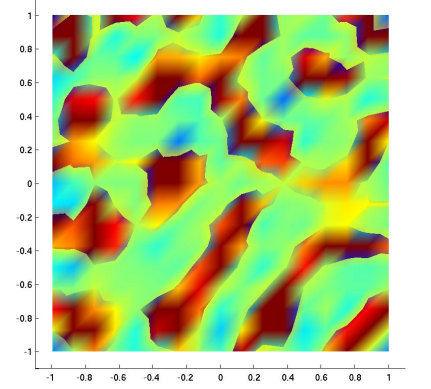
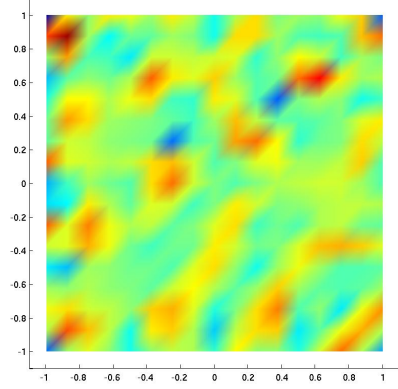


4. u and v for $timestep = 0.1$, $time = 50$ and 16641 elements in the mesh:

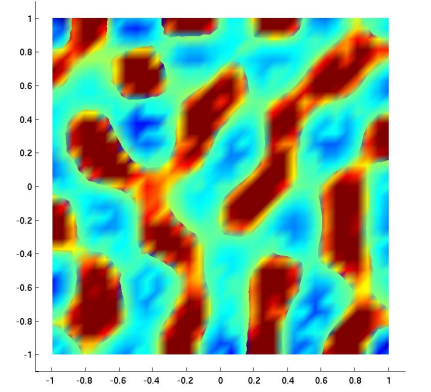
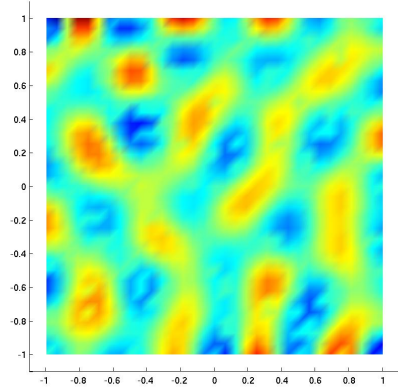


For the next tests we set a new time step.

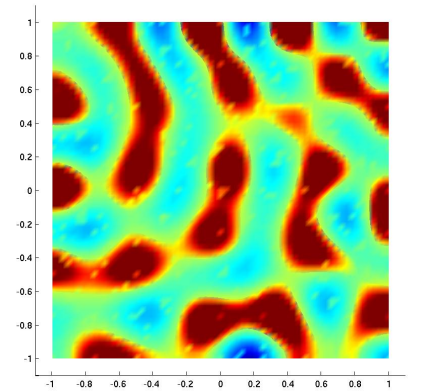
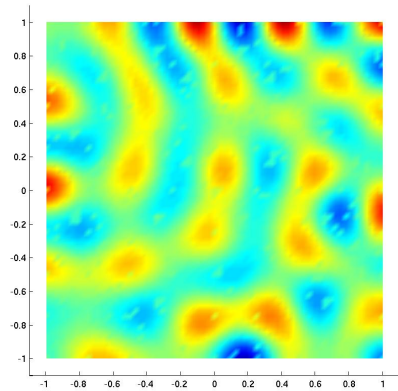
1. u and v for $timestep = 0.01$, $time = 50$ and 289 elements in the mesh:



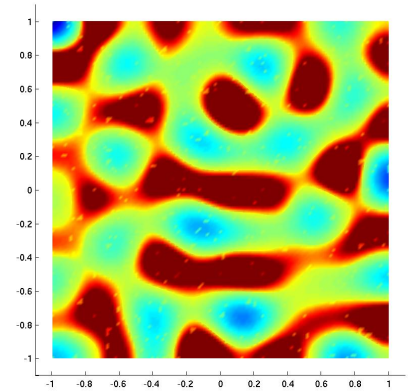
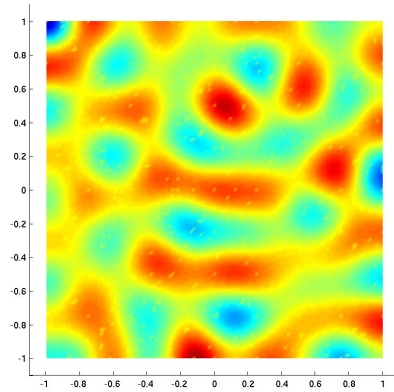
2. u and v for $timestep = 0.01$, $time = 50$ and 1089 elements in the mesh:



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:

```

1 function [ u0, v0 ] = InitialCondition( N )
2
3 u0 = zeros( round(sqrt(N)) );
4 v0 = zeros( round(sqrt(N)) );
5
6 u0(:, :) = 0.005;
7 v0(:, :) = 0.0005;
8
9 for i = 1:2:( round(sqrt(N)) )
10     u0(i, :) = 0.01;
11     v0(i, :) = 0.001;
12 end
13
14 end

```

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