```matlab
function fe2dx_nr_fast ( alpha, beta, gamma, delta, T, delt, u0f, v0f, ...
  k1, k2, g2uf, g2vf )
%*****************************************************************************80
%
%% FE2DX_NR_FAST applies Scheme 1 with Kinetics 1 to predator prey in a region.
%
%  Discussion:
%
%    FE2DX_NR_FAST is a "fast" version of FE2DX_NR.
%
%    FE2DX_NR is a finite element Matlab code for Scheme 1 applied
%    to the predator-prey system with Kinetics 1 solved over a region
%    which has been triangulated.  The geometry and grid are read from
%    user-supplied files 't_triang.dat' and 'p_coord.dat' respectively,
%    as are the list of nodes on which Robin and Neumann boundary
%    conditions are to be imposed (from 'bn1_nodes.dat' and 'bn2_nodes.dat'
%    respectively).
%
%    This function has 12 input parameters.  All, some, or none of them may
%    be supplied as command line arguments or as functional parameters.
%    Parameters not supplied through the argument list will be prompted for.
%
%    The parameters ALPHA, BETA, GAMMA and DELTA appear in the predator-prey
%    equations as follows:
%
%      dUdT =         nabla U +      U*V/(U+ALPHA) + U*(1-U)
%      dVdT = delta * nabla V + BETA*U*V/(U+ALPHA) - GAMMA * V
%
%  Licensing:
%
%    Copyright (C) 2014 Marcus R. Garvie.
%    See 'mycopyright.txt' for details.
%
%  Modified:
%
%    29 April 2014
%
%  Author:
%
%    Marcus R. Garvie and John Burkardt.
%
%  Reference:
%
%    Marcus R Garvie, John Burkardt, Jeff Morgan,
%    Simple Finite Element Methods for Approximating Predator-Prey Dynamics
%    in Two Dimensions using MATLAB,
%    Submitted to Bulletin of Mathematical Biology, 2014.
%
%  Parameters:
%
%    Input, real ALPHA, a parameter in the predator prey equations.
%    0 < ALPHA.
%
%    Input, real BETA, a parameter in the predator prey equations.
%    0 < BETA.
```

```
%
%    Input, real GAMMA, a parameter in the predator prey equations.
%    0 < GAMMA.
%
%    Input, real DELTA, a parameter in the predator prey equations.
%    0 < DELTA.
%
%    Input, real T, the maximum time.
%    0 < T.
%
%    Input, real DELT, the time step to use in integrating from 0 to T.
%    0 < DELT.
%
%    Input, string U0F or function pointer @U0F, a function for the initial
%    condition of U(X,Y).
%
%    Input, string V0F or function pointer @V0F, a function for the initial
%    condition of V(X,Y).
%
%    Input, real K1, the coefficient for the Robin boundary condition
%    to be applied to U: dU/dn = k1 * U.
%
%    Input, real K2, the coefficient for the Robin boundary condition
%    to be applied to V: dV/dn = k2 * V.
%
%    Input, string G2UF or function pointer @G2UF, a function for the Neumann
%    boundary condition of U(X,Y,T).
%
%    Input, string G2VF or function pointer @G2VF, a function for the Neumann
%    boundary condition of V(X,Y,T).
%
%*****************************************************************************80
%  Enter data for mesh geometry.
%*****************************************************************************80
%
%  Read in 'p(2,n)', the 'n' coordinates of the nodes.
%
  load p_coord.dat -ascii
  p = ( p_coord )';
%
%  Read in 't(3,no_elems)', the list of nodes for 'no_elems' elements.
%
  load t_triang.dat -ascii
  t = ( round ( t_triang ) )';
%
%  Read in 'bn1(1,isn1)', the nodes on Gamma1.
%
  load bn1_nodes.dat -ascii
  bn1 = ( round ( bn1_nodes ) )';
%
%  Read in 'bn2(1,isn2)', the nodes on Gamma2.
%
  load bn2_nodes.dat -ascii
  bn2 = ( round ( bn2_nodes ) )';
%
%  Construct the connectivity for the nodes on Gamma1.
%
```

```
cpp1 = subsetconnectivity ( t', p', bn1' );
%
%  Construct the connectivity for the nodes on Gamma2.
%
cpp2 = subsetconnectivity ( t', p', bn2' );
%
%  E1 = number of edges on Gamma1.
%
[ e1, ~ ] = size ( cpp1 );
%
%  E2 = number of edges on Gamma2.
%
[ e2, ~ ] = size ( cpp2 );
%
%  N = degrees of freedom per variable.
%
[ ~, n ] = size ( p );
%
%  NO_ELEMS = number of elements.
%
[ ~, no_elems ] = size ( t );
%
%  Extract vector of 'x' and 'y' values.
%
x = p(1,:);
y = p(2,:);
%*****************************************************************************80
%  Enter data for model.
%*****************************************************************************80
if ( nargin < 1 )
  alpha = input ( 'Enter parameter alpha:  ' );
elseif ( ischar ( alpha ) )
  alpha = str2num ( alpha );
end
if ( nargin < 2 )
  beta = input ( 'Enter parameter beta:  ' );
elseif ( ischar ( beta ) )
  beta = str2num ( beta );
end
if ( nargin < 3 )
  gamma = input ( 'Enter parameter gamma:  ' );
elseif ( ischar ( gamma ) )
  gamma = str2num ( gamma );
end
if ( nargin < 4 )
  delta = input ( 'Enter parameter delta:  ' );
elseif ( ischar ( delta ) )
  delta = str2num ( delta );
end
if ( nargin < 5 )
  T = input ( 'Enter maximum time T:  ' );
elseif ( ischar ( T ) )
  T = str2num ( T );
end
if ( nargin < 6 )
  delt = input ( 'Enter time-step delt:  ' );
elseif ( ischar ( delt ) )
```

```
    delt = str2num ( delt );
  end
  fprintf ( 1, '  Using ALPHA = %g\n', alpha );
  fprintf ( 1, '  Using BETA = %g\n', beta );
  fprintf ( 1, '  Using GAMMA = %g\n', gamma );
  fprintf ( 1, '  Using DELTA = %g\n', delta );
  fprintf ( 1, '  Using T = %g\n', T );
  fprintf ( 1, '  Using DELT = %g\n', delt );
%
%  Initial conditions.
%
  if ( nargin < 7 )
    u0_str = input ( 'Enter initial data function u0(x,y):  ', 's' );
    u0f = @(x,y) eval ( u0_str );
  elseif ( ischar ( u0f ) )
    u0_str = u0f;
    u0f = @(x,y) eval ( u0_str );
  end
  u = ( arrayfun ( u0f, x, y ) )';
  if ( nargin < 8 )
    v0_str = input ( 'Enter initial data function v0(x,y):  ', 's' );
    v0f = @(x,y) eval ( v0_str );
  elseif ( ischar ( v0f ) )
    v0_str = v0f;
    v0f = @(x,y) eval ( v0_str );
  end
  v = ( arrayfun ( v0f, x, y ) )';
%
%  Boundary conditions.
%
  if ( nargin < 9 )
    k1 = input('Enter the parameter k1 in the Robin b.c. for u  ');
  elseif ( ischar ( k1 ) )
    k1 = str2num ( k1 );
  end
  if ( nargin < 10 )
    k2 = input('Enter the parameter k2 in the Robin b.c. for v  ');
  elseif ( ischar ( k2 ) )
    k2 = str2num ( k2 );
  end
  if ( nargin < 11 )
    g2u_str = input('Enter the Neumann b.c. g2u(x,y,t) for u  ','s');
    g2uf = @(x,y,t)eval(g2u_str);
  elseif ( ischar ( g2uf ) )
    g2u_str = g2uf;
    g2uf = @(x,y,t) eval ( g2u_str );
  end
  if ( nargin < 12 )
    g2v_str = input('Enter the Neumann b.c. g2v(x,y,t) for v  ','s');
    g2vf = @(x,y,t)eval(g2v_str);
  elseif ( ischar ( g2vf ) )
    g2v_str = g2vf;
    g2vf = @(x,y,t) eval ( g2v_str );
  end
%
%  N = number of time steps.
%
```

```
N = round ( T / delt );
fprintf ( 1, '  Taking N = %d time steps\n', N );
%*****************************************************************************80
%  Assembly.
%*****************************************************************************80
m_hat = zeros(n,1);
K = sparse ( n, n );
for elem = 1 : no_elems
%
%  Identify nodes ni, nj and nk in element 'elem'.
%
  ni = t(1,elem);
  nj = t(2,elem);
  nk = t(3,elem);
%
%  Identify coordinates of nodes ni, nj and nk.
%
  xi = p(1,ni);
  xj = p(1,nj);
  xk = p(1,nk);
  yi = p(2,ni);
  yj = p(2,nj);
  yk = p(2,nk);
%
%  Calculate the area of element 'elem'.
%
  triangle_area = abs(xj*yk-xk*yj-xi*yk+xk*yi+xi*yj-xj*yi)/2;
%
%  Calculate some quantities needed to construct elements in K.
%
  h1 = (xi-xj)*(yk-yj)-(xk-xj)*(yi-yj);
  h2 = (xj-xk)*(yi-yk)-(xi-xk)*(yj-yk);
  h3 = (xk-xi)*(yj-yi)-(xj-xi)*(yk-yi);
  s1 = (yj-yi)*(yk-yj)+(xi-xj)*(xj-xk);
  s2 = (yj-yi)*(yi-yk)+(xi-xj)*(xk-xi);
  s3 = (yk-yj)*(yi-yk)+(xj-xk)*(xk-xi);
  t1 = (yj-yi)^2+(xi-xj)^2;
  t2 = (yk-yj)^2+(xj-xk)^2;
  t3 = (yi-yk)^2+(xk-xi)^2;
%
%  Calculate local contributions to m_hat.
%
  m_hat_i = triangle_area/3;
  m_hat_j = m_hat_i;
  m_hat_k = m_hat_i;
%
%  Calculate local contributions to K.
%
  K_ki = triangle_area*s1/(h3*h1);
  K_ik = K_ki;
  K_kj = triangle_area*s2/(h3*h2);
  K_jk = K_kj;
  K_kk = triangle_area*t1/(h3^2);
  K_ij = triangle_area*s3/(h1*h2);
  K_ji = K_ij;
  K_ii = triangle_area*t2/(h1^2);
  K_jj = triangle_area*t3/(h2^2);
```

```
%
%  Add contributions to vector m_hat.
%
    m_hat(nk)=m_hat(nk)+m_hat_k;
    m_hat(nj)=m_hat(nj)+m_hat_j;
    m_hat(ni)=m_hat(ni)+m_hat_i;
%
%  Add contributions to K.
%
    K=K+sparse(nk,ni,K_ki,n,n);
    K=K+sparse(ni,nk,K_ik,n,n);
    K=K+sparse(nk,nj,K_kj,n,n);
    K=K+sparse(nj,nk,K_jk,n,n);
    K=K+sparse(nk,nk,K_kk,n,n);
    K=K+sparse(ni,nj,K_ij,n,n);
    K=K+sparse(nj,ni,K_ji,n,n);
    K=K+sparse(ni,ni,K_ii,n,n);
    K=K+sparse(nj,nj,K_jj,n,n);
  end
%
%  Construct matrix L.
%
  ivec = 1:n;
  IM_hat = sparse(ivec,ivec,1./m_hat,n,n);
  L = delt * IM_hat * K;
%
%  Construct fixed parts of matrices A_{n-1} and C_{n-1}.
%
  A0 = L + sparse(1:n,1:n,1-delt,n,n);
  C0 = delta * L + sparse(1:n,1:n,1+delt*gamma,n,n);
%*****************************************************************************80
%  Time-stepping.
%*****************************************************************************80
  for nt = 1 : N
    tn = nt * delt;
%
%  Initialize right-hand-side functions.
%
    rhs_u = u;
    rhs_v = v;
%
%  Update coefficient matrices of linear system.
%
    diag = abs ( u );
    diag_entries = u ./ ( alpha + abs ( u ) );
    A = A0 +         delt * sparse(1:n,1:n,diag,n,n);
    B =              delt * sparse(1:n,1:n,diag_entries,n,n);
    C = C0 - beta * delt * sparse(1:n,1:n,diag_entries,n,n);
%
%  Do the incomplete LU factorisation of C and A.
%
    [ LC, UC ] = ilu ( C, struct('type','ilutp','droptol',1e-5) );
    [ LA, UA ] = ilu ( A, struct('type','ilutp','droptol',1e-5) );
%
%  Impose Robin boundary condition on Gamma1.
%
    for i = 1:e1
```

```
      node1 = cpp1(i,1);
      node2 = cpp1(i,2);
      x1 = p(1,node1);
      y1 = p(2,node1);
      x2 = p(1,node2);
      y2 = p(2,node2);
      im_hat1 = 1/m_hat(node1);
      im_hat2 = 1/m_hat(node2);
      gamma12 = sqrt((x1-x2)^2 + (y1-y2)^2);
      rhs_u(node1) = rhs_u(node1) + delt*k1*u(node1)*im_hat1*gamma12/2;
      rhs_u(node2) = rhs_u(node2) + delt*k1*u(node2)*im_hat2*gamma12/2;
      rhs_v(node1) = rhs_v(node1) + delt*k2*v(node1)*im_hat1*gamma12/2;
      rhs_v(node2) = rhs_v(node2) + delt*k2*v(node2)*im_hat2*gamma12/2;
    end
%
%  Impose Neumann boundary condition on Gamma2.
%
    for i = 1:e2
      node1 = cpp2(i,1);
      node2 = cpp2(i,2);
      x1 = p(1,node1);
      y1 = p(2,node1);
      x2 = p(1,node2);
      y2 = p(2,node2);
      im_hat1 = 1/m_hat(node1);
      im_hat2 = 1/m_hat(node2);
      gamma12 = sqrt((x1-x2)^2 + (y1-y2)^2);
      rhs_u(node1) = rhs_u(node1) + delt * g2uf (x1,y1,tn) * im_hat1*gamma12/2;
      rhs_u(node2) = rhs_u(node2) + delt * g2uf (x2,y2,tn) * im_hat2*gamma12/2;
      rhs_v(node1) = rhs_v(node1) + delt * g2vf (x1,y1,tn) * im_hat1*gamma12/2;
      rhs_v(node2) = rhs_v(node2) + delt * g2vf (x2,y2,tn) * im_hat2*gamma12/2;
    end
%
%  Solve for v using GMRES.
%
    [v,flagv,relresv,iterv] = gmres ( C,rhs_v,[],1e-6,[],LC,UC,v );
    if flagv ~= 0
      flagv
      relresv
      iterv
      error('GMRES did not converge')
    end
    r = rhs_u - B * v;
%
%  Solve for u using GMRES.
%
    [u,flagu,relresu,iteru] = gmres ( A,r,[],1e-6,[],LA,UA,u );
    if flagu ~= 0
      flagu
      relresu
      iteru
      error('GMRES did not converge')
    end
  end
%**********************************************************************80
%  Plot the solutions.
%**********************************************************************80
```

```
%
%  Plot U;
%
  figure;
  set(gcf,'Renderer','zbuffer');
  trisurf(t',x,y,u,'FaceColor','interp','EdgeColor','interp');
  colorbar;
  axis off;
  title('u');
  view ( 2 );
  axis equal on tight;
  filename = 'fe2dx_nr_fast_u.png';
  print ( '-dpng', filename );
  fprintf ( 1, '  Saved graphics file "%s"\n', filename );
%
%  Plot V.
%
  figure;
  set(gcf,'Renderer','zbuffer');
  trisurf(t',x,y,v,'FaceColor','interp','EdgeColor','interp');
  colorbar;
  axis off;
  title('v');
  view ( 2 );
  axis equal on tight;
  filename = 'fe2dx_nr_fast_v.png';
  fprintf ( 1, '  Saved graphics file "%s"\n', filename );
  print ( '-dpng', filename );
  return
end
```

Published with MATLAB® R2013b