```matlab
function out=wave_solve(c,L,n,sigma,T,M,u0,method)

%
% --inputs--
% c:        advective speed
% L:        domain size [0,L]
% n:        number of interior grid points
% sigma:    Courant number
% T:        final time
% M:        number of solutions recorded between [0,T]
% u0:       function that prescribes the initial conditions u0(x)
% method:   integration method, one of:
%           'forward-upwind'
%           'implicit-central'
%           'beam-warming'
%           'lax-wendroff'
%         ...
%
% --outputs--
% out.h   grid spacing
% out.k   time step size
% out.l   number of time steps taken
% out.x:  spatial locations so that out.x(1)=0 and out.x(end)=L
% out.TT: out.TT(1)=0 and out.TT(end)=T with
%         length(out.TT)=M+2;
% out.U:  numerical solution as matrix
%         out.U(:,j) is the numerical solution at time out.TT(j)
%         with j=1,\dots,M+2
%         size(out.U,1)=length(out.x)
%         size(out.U,2)=length(out.TT)
%

% set output to empty
out=[];

% work on grid
h=L/(n+1); % grid spacing recovered from the number of interior points
out.h = h; % store it

out.x=[0:h:L]; % actual grid array, including x=0 and x=L
N=length(out.x); % number of overall points

% time outputs
out.TT=linspace(0,T,M+2);

% build the matrix for the updates
switch lower(method)

case 'forward-upwind'

  if ( c<0 )
      error('please specify a positive advective speed');
  end

  A = -diag(ones(N,1),0) - ...
      -diag(ones(N-1,1),-1);
  A(1,n+1)=1; % periodic boundary on U(1)=U_0

case 'implicit-central'
  if ( c<0 )
      error('please specify a positive advective speed');
  end

  A = diag(zeros(N,1),0) + diag(ones(N-1,1),1) - diag(ones(N-1,1),-1);
  % create a tridiagonal matrix with -1, 0, 1
  A = .5*A;
  C = diag(ones(N,1),0); % create a diagonal matrix with 1

  % we have to create two separate matrices for implicit central due to it
  % being an implicit method


  % set boundary conditions based on finite difference method equation
  A(1,N-1) = -.5;
  A(N,2) = .5;


case 'beam-warming'
  if ( c<0 )
      error('please specify a positive advective speed');
  end


  A = zeros(N) + diag(ones(N,1),0) * 3 + diag(ones(N-1,1),-1) * -4 + ...
      diag(ones(N-2,1),-2);
  % create a slightly shifted tridiagonal matrix with 1,-4,3
  A(1,N-2) = 1;
  A(1,N-1) = -4;
  A(2,N-1) = 1;
  % set boundary conditions based on finite difference method equation

  B = zeros(N) + diag(ones(N,1),0) * 1 + diag(ones(N-1,1),-1) * -2 + ...
      diag(ones(N-2,1),-2);
  B(1,N-2) = 1;
  B(1,N-1) = -2;
  B(2,N-1) = 1;
  % create a second tridiagonal matrix. We have to do this due to the
  % nature of the beam warming equation having different coefficients




case 'lax-wendroff'
  % create a tridiagonal matrix with -1,0,1
  A = zeros(N) + diag(ones(N-1,1),1) - diag(ones(N-1,1),-1);
  A(1,N-1) = -1;
  A(N,2) = 1;

  % tridiagonal matrix with 1,-2,1
  B = zeros(N) -2* diag(ones(N,1),0) + diag(ones(N-1,1),1) + ...
      diag(ones(N-1,1),-1);
  B(1,N-1) = 1;
  B(N,2) = 1;



otherwise
  error('method is unknown');
end

% time step size recovered from Courant number
k=sigma*h/c;
out.k = k; % store it

% initial conditions
U_=u0(out.x)'; t=0; j=1;

% store initial conditions
out.U(:,j)=U_; j=j+1;

% integrate in time
l=0;
while t<out.TT(end)
```

```
   % pick the smallest between the time step
   % and the time step to get to the next out.TT(j)
   k_=min([k,(out.TT(j)-t)]);
   sigma_=k_*c/h;

   fprintf('Time: %f; Sigma = %f; Time step = %f\n',t,sigma_,k_);

   % zero the update
   dU_=zeros(size(U_));

   switch lower(method)

   case {'forward-upwind'}

      dU_=sigma_*A*U_; % Euler fwd step

   case {'implicit-central'}

      u_next = ((sigma_*A)+C) \ (U_);
      dU_ = -sigma_*A*u_next;
      % we have to use a temporary variable u_next due to the implicit
      % nature of this method. First we must compute our coefficient matrix
      % divided by U_, then we plug in the result into U_

   case 'beam-warming'

      dU_ = (-(sigma_ / 2) * A + (sigma_^2 / 2) * B) * U_;
      % multiply the coefficient matrices by sigma/2, then multiply by U_
      % to find the majority of the rhs of the beam warming equation

   case 'lax-wendroff'

      dU_ = (-(sigma_ / 2) * A + (sigma_^2 / 2) * B) * U_;
      % multiply the coefficient matrices by sigma/2, sigma^2/2 then
      % multiply by U_ to find the majority of the rhs of the lax-wendroff
      % equation

   otherwise
     error('method is unknown');
   end

   % update
   U_=U_+dU_;

   % advance time to reflect update
   t=t+k_;
   l=l+1; % step counter

   % store
   if ( t==out.TT(j) )
     out.TT(j)=t; % adjust recorded time
     out.U(:,j)=U_; j=j+1;
   end

end

out.l=l; % number of steps
```

```
Not enough input arguments.

Error in wave_solve (line 37)
h=L/(n+1); % grid spacing recovered from the number of interior points
```