```
function [V,FD_grid,time_steps] =
 vega_timestepping(sigma,r,rhs,bc_right,init_cond,S_max,NS,T,Nt,theta)
% vega_timestepping
%
% Inputs:
%   sigma       : volatility (real number)
%   r           : risk-free return (real number)
%   rhs         : matrix of values for the right-hand-side of the
 equation
%                   dimension (NS+1,Nt+1)
%                   rows = spatial nodes, columns = time steps
%   bc_right    : @-function describing the boundary condition on the
 right border, bc_right = @(t) ...
%                   t is a real number
%   init_cond   : @-function describing the initial condition of the
 problem, initial_cond = @(S) ...
%                   S is a vector (spatial grid)
%   S_max       : real value setting the upper extreme of the interval
 in which the solution is computed
%   NS          : number of intervals in the discretization of
 [0,Rmax]
%   T           : final time of the equation
%   Nt          : number of time steps
%   theta       : parameter for the theta-method time-stepping scheme:
%                   theta = 0   --> Forward Euler
%                   theta = 1   --> Backward Euler
%                   theta = 0.5 --> Crank-Nicholson
%
% Outputs:
%   V           : matrix containing the solution of the PDE
%                   rows = spatial nodes, columns = time steps
%   grid        : spatial grid, contains the nodes on which the
 solution H is evaluated
%   time_steps  : time grid, contains the time steps at which H is
 computed

% set grid and grid-size
h = S_max/NS;
FD_grid = linspace(0,S_max,NS+1);
inner_grid = FD_grid(1:end-1);

% set time steps
dt = T/Nt;
time_steps = dt*(0:Nt);

% initialize solution matrix
V = zeros(NS+1,Nt+1);

% set initial condition
V(:,1) = init_cond(FD_grid)';
V(end,:) = bc_right(time_steps);
```

```
% define auxiliary vectors
%   for reading convenience and for computational efficiency
S = inner_grid';
Ssq = inner_grid'.^2;

% build matrix A
%   space-dependent but not time dependent
eta = sigma^2/h^2*Ssq;
nu = 0.5*r*S/h;
A_up = -0.5*eta(1:end-1) - nu(1:end-1);
A_main = eta + r;
A_down = -0.5*eta(2:end) + nu(2:end);

% correction term for vector f
corr_right = 0.5*sigma^2/h^2*S_max^2 + 0.5*r*S_max/h;

% initialize time
t_old = 0;

% build f_old with boundary corrections, evaluated at t=0
f_old = rhs(1:end-1,1);
f_old(end) = f_old(end) + corr_right*bc_right(t_old);

% Main time-stepping loop

for tn = 1:Nt

    % current time
    t_new = t_old + dt;

    % built f_new with boundary corrections
    f_new = rhs(1:end-1,tn+1);
    f_new(end) = f_new(end) + corr_right*bc_right(t_new);

    % the system evolves according to the formula
    %   (I + dt theta A) V_new = (I - dt (1-theta) A) V_old + dt f_tot
    % where f_tot = (1-theta) f_old + theta f_new
    % Define:
    %   X = (I + dt theta A)
    %   Y = (I - dt (1-theta) A)

    % build matrix X
    X_up = dt*theta*A_up;
    X_main = 1 + dt*theta*A_main;
    X_down = dt*theta*A_down;

    % build matrix Y
    Y_up = -dt*theta*A_up;
    Y_main = 1 - dt*theta*A_main;
    Y_down = -dt*theta*A_down;
    Y = spdiags([0;Y_up],1,NS,NS) + spdiags(Y_main,0,NS,NS) +...
        spdiags([Y_down;0],-1,NS,NS);

    % build vector f_tot
```

```
    f_tot = (1-theta)*f_old + theta*f_new;

    % solve the system and store the solution
    RHS = Y*V(1:end-1,tn) + dt*f_tot;
    V(1:end-1,tn+1) = thomas(X_down,X_main,X_up,RHS);

    % update current time
    f_old = f_new;

end

end
```

```
Not enough input arguments.

Error in vega_timestepping (line 30)
h = S_max/NS;
```

Published with MATLAB® R2020a