7

The Korteweg–deVries Equation

This partial differential equation (PDE) problem introduces the following mathematical concepts and computational methods:

1. A nonlinear PDE with an exact solution that can be used to assess the accuracy of a numerical method of lines (MOL) solution.
2. Some of the basic properties of *solitons* that have broad application in many areas of physics, engineering, and the biological sciences.
3. Nonlinear interaction of two (or more) solitons.
4. Illustrative programming of a soliton solution.
5. Sparse matrix integration of the ordinary differential equations (ODEs) for the MOL approximation of a PDE.
6. Computer analysis of a PDE over an essentially infinite spatial domain.
7. Evaluation of *invariants of the solution* (integrals of functions of the solution that do not change with t).
8. Comparison of procedural and vectorized programming in Matlab.

The *one-dimensional (1D) Korteweg–deVries equation (KdV) equation* is [1]

$$\frac{\partial u}{\partial t} + 6u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} = 0$$

or in subscript notation,

$$u_t + 6uu_x + u_{xxx} = 0 \tag{7.1}$$

where

u dependent variable
x boundary-value (spatial) independent variable
t initial-value independent variable

Equation (7.1) is the conventional, nondimensional version of the equation originally derived by Korteweg and de Vries [2] (some background information is given in Appendix C). It is first order in t and third order in x, and therefore requires one

initial condition (IC) and three *boundary conditions* (BCs). The IC is taken from the analytical solution (for a single soliton with velocity c)

$$u(x,t) = \frac{1}{2}c\,\mathrm{sech}^2\left\{\frac{1}{2}\sqrt{c}(x-ct)\right\} \tag{7.2}$$

with $t = 0$; that is,

$$u(x,t=0) = \frac{1}{2}c\,\mathrm{sech}^2\left\{\frac{1}{2}\sqrt{c}(x)\right\} \tag{7.3}$$

To complete the specification of the problem, we would naturally consider the three required BCs for Eq. (7.1). However, if the PDE is analyzed over an essentially infinite domain, $-\infty \le x \le \infty$, and if changes in the solution occur only over a finite interval in x, then *BCs at infinity have no effect*; in other words, we do not have to actually specify BCs (since they have no effect). This situation will be clarified through the PDE solution.

Consequently, Eqs. (7.1) and (7.3) constitute the complete PDE problem. The solution to this problem, Eq. (7.2), is used in the subsequent programming and analysis to evaluate the numerical MOL solution. Note that Eq. (7.2) is a *traveling wave* solution since the argument of the sech function is $x - ct$.

We now consider some Matlab routines for a numerical MOL solution of Eqs. (7.1) and (7.3) with the analytical solution, Eq. (7.2), included. A main program, `pde_1_main`, is given in Listing 7.1.

```
%
% Clear previous files
  clear all
  clc
%
% Parameters shared with other routines
  global  ncall ncase    c    c1    c2    xl    xu     x     n
%
% Select case; 1 - one soliton; 2 - two solitons;
  ncase=1;
  if(ncase==1)c=1.0; n=201; end
  if(ncase==2)c1=2.0; c2=0.5; n=301; end
%
% Initial condition
  t0=0.0;
  u0=inital_1(t0);
%
% Independent variable for ODE integration
  tf=30.0;
  tout=[t0:10.0:tf]';
  nout=4;
  ncall=0;
%
```

```
% ODE integration
  mf=2;
  reltol=1.0e-06; abstol=1.0e-06;
  options=odeset('RelTol',reltol,'AbsTol',abstol);
%
% Explicit (nonstiff) integration
  if(mf==1)[t,u]=ode45(@pde_1,tout,u0,options); end
%
% Implicit (sparse stiff) integration
  if(mf==2)
    S=jpattern_num;
    pause
    options=odeset(options,'JPattern',S)
    [t,u]=ode15s(@pde_1,tout,u0,options);
  end
%
% Store analytical solution, errors in numerical solution
  if(ncase==1)
    for it=1:nout
      for i=1:n
        u_anal(it,i)=ua(x(i),t(it));
        err(it,i)=u(it,i)-u_anal(it,i);
      end
    end
%
%   Display selected output
    fprintf('\n ncase = %2d   c = %5.2f\n',ncase,c);
    for it=1:nout
      fprintf('     t       x        u(it,i) u_anal(it,i)
              err(it,i)\n');
      for i=1:5:n
        fprintf('%6.2f%8.3f%15.6f%15.6f%15.6f\n',...
              t(it),x(i),u(it,i),u_anal(it,i),err(it,i));
      end
%
%     Calculate and display three invariants
      ui=u(it,:);
      uint=simp(xl,xu,n,ui);
      fprintf('\n Invariants at t = %5.2f',t(it));
      fprintf('\n    I1 = %10.4f   Mass conservation'   , ...
              uint(1));
      fprintf('\n    I2 = %10.4f   Energy conservation' , ...
              uint(2));
      fprintf('\n    I3 = %10.4f   Whitham invariant\n\n', ...
              uint(3));
    end
    fprintf('    ncall = %4d\n\n',ncall);
```

```
%
%   Plot numerical and analytical solutions
    plot(x,u,'o',x,u_anal,'-')
    xlabel('x')
    ylabel('u(x,t)')
    title('KdV equation; t = 0, 10, 20, 30; o - numerical;
          solid - analytical')
    print -deps -r300 pde2.eps; print -dps -r300 pde2.ps;
    print -dpng -r300 pde2.png
  end
%
% Store numerical solution
  if(ncase==2)
%
%   Display selected output
    fprintf('\n ncase = %2d  c1 = %5.2f  c2 = %5.2f\n', ...
           ncase,c1,c2);
    for it=1:nout
      fprintf('     t       x        u(it,i)\n');
      for i=1:5:n
        fprintf('%6.2f%8.3f%15.6f\n',t(it),x(i),u(it,i));
      end
      fprintf('\n');
%
%     Calculate and display three invariants
      ui=u(it,:);
      uint=simp(xl,xu,n,ui);
      fprintf('\n Invariants at t = %5.2f',t(it));
      fprintf('\n    I1 = %10.4f   Mass conservation'   , ...
              uint(1));
      fprintf('\n    I2 = %10.4f   Energy conservation' , ...
              uint(2));
      fprintf('\n    I3 = %10.4f   Whitham invariant\n\n', ...
              uint(3));
    end
    fprintf('  ncall = %4d\n\n',ncall);
%
%   Plot numerical solution
    for it=1:nout
      subplot(2,2,it)
      plot(x,u(it,:),'-')
      axis([-30 70 0 1])
      xlabel('x')
      if(it==1)
        ylabel('u(x,0)')
        title('KdV; t = 0')
      elseif(it==2)
```

```
      ylabel('u(x,10)')
      title('KdV; t = 10')
    elseif(it==3)
      ylabel('u(x,20)')
      title('KdV; t = 20')
    elseif(it==4)
      ylabel('u(x,30)')
      title('KdV; t = 30')
    end
   end
end
```

Listing 7.1. Main program pde_1_main

We can note the following points about this program:

1. After specifying some *global* variables, the program executes one of two cases with ncase=1 for one soliton and ncase=2 for two solitons. Note that for ncase=1, c=1 and $n = 201$ grid points in x are used, while for ncase=2, two soliton velocities are specified, c1=2, c2=0.5, and $n = 301$ grid points in x are used since the two solitons require greater spatial resolution (larger n) than for the single-soliton case (as will be demonstrated subsequently).

```
%
% Clear previous files
  clear all
  clc
%
% Parameters shared with other routines
  global  ncall ncase     c    c1    c2    xl    xu    x    n
%
% Select case; 1 - one soliton; 2 - two solitons;
  ncase=1;
  if(ncase==1)c=1.0; n=201; end
  if(ncase==2)c1=2.0; c2=0.5; n=301; end
```

2. The IC, Eq. (7.3), is defined by a call to routine inital_1 that defines the IC over the interval $-30 \leq x \leq 70$ for either one solton (ncase=1) or two solitons (ncase=2), as explained subsequently. The t interval is then defined as $0 \leq t \leq 30$ with solution outputs at $t = 0, 10, 20, 30$.

```
%
% Initial condition
  t0=0.0;
  u0=inital_1(t0);
```

```
%
% Independent variable for ODE integration
  tf=30.0;
  tout=[t0:10.0:tf]';
  nout=4;
  ncall=0;
```

3. The n ODEs are integrated by a call to either a nonstiff integrator, `ode45` (for `mf=1`), or a stiff integrator, `ode15s` (for `mf=2`). The stiff integrator is recommended since the nonstiff integrator requires rather long computer runs, typically an indication of stiffness. The MOL ODE routine is `pde_1` (discussed subsequently).

   Note also that the stiff integrator is run in a sparse matrix mode, as reflected in the second `options` command. This call to `ode15s` illustrates a general approach to sparse matrix ODE integration, which is a particularly effective approach in the MOL analysis of PDE systems.

```
%
% ODE integration
  mf=2;
  reltol=1.0e-06; abstol=1.0e-06;
  options=odeset('RelTol',reltol,'AbsTol',abstol);
%
% Explicit (nonstiff) integration
  if(mf==1)[t,u]=ode45(@pde_1,tout,u0,options); end
%
% Implicit (sparse stiff) integration
  if(mf==2)
    S=jpattern_num;
    pause
    options=odeset(options,'JPattern',S)
    [t,u]=ode15s(@pde_1,tout,u0,options);
  end
```

4. For the single-soliton case (`ncase=1`), the analytical solution of Eq. (7.2) is then evaluated by a call to `ua` (discussed subsequently) and selected numerical output for the numerical and analytical solutions is displayed by the two nested `for` loops (the first for t and the second for x).

```
%
% Store analytical solution, errors in numerical solution
  if(ncase==1)
    for it=1:nout
      for i=1:n
```

```
        u_anal(it,i)=ua(x(i),t(it));
        err(it,i)=u(it,i)-u_anal(it,i);
      end
    end
%
%   Display selected output
    fprintf('\n ncase = %2d   c = %5.2f\n',ncase,c);
    for it=1:nout
      fprintf('     t       x          u(it,i)   u_anal(it,i)
              err(it,i)\n');
      for i=1:5:n
        fprintf('%6.2f%8.3f%15.6f%15.6f%15.6f\n',...
              t(it),x(i),u(it,i),u_anal(it,i),err(it,i));
      end
```

5. Three invariants are evaluated by a call to `simp` that implements a numerical quadrature (integration) based on Simpson's rule (discussed subsequently).

```
%
%   Calculate and display three invariants
    ui=u(it,:);
    uint=simp(xl,xu,n,ui);
    fprintf('\n Invariants at t = %5.2f',t(it));
    fprintf('\n    I1 = %10.4f   Mass conservation'    , ...
            uint(1));
    fprintf('\n    I2 = %10.4f   Energy conservation'  , ...
            uint(2));
    fprintf('\n    I3 = %10.4f   Whitham invariant\n\n', ...
            uint(3));
  end
  fprintf('    ncall = %4d\n\n',ncall);
```

The integral invariants are

1. Conservation of mass:

$$u_1(t) = \int_{-\infty}^{\infty} u(x,t)dx \tag{7.4a}$$

2. Conservation of energy:

$$u_2(t) = \int_{-\infty}^{\infty} \frac{1}{2}u^2(x,t)dx \tag{7.4b}$$

3. Proposed by Whitham [3]:

$$u_3(t) = \int_{-\infty}^{\infty} 2u^3(x,t) - u_x^2(x,t)dx \tag{7.4c}$$

The exact values of these invariants for the case of Eq. (7.2) are $u_1(t) = 2, u_2(t) = 1/3,$ and $u_3(t) = 0.4$, which are subsequently compared with the values computed by numerical integration from the routine `simp`.

6. The numerical and analytical solutions are then plotted.

```
%
%   Plot numerical and analytical solutions
    plot(x,u,'o',x,u_anal,'-')
    xlabel('x')
    ylabel('u(x,t)')
    title('KdV equation; t = 0, 10, 20, 30;
          o - numerical; solid - analytical')
  end
```

Selected numerical output and the plotted output for `ncase=1` are given in Table 7.1.

Table 7.1. Partial output from `pde_1_main` and `pde_1` (`ncase=1`)

```
ncase =  1   c =  1.00

    t       x        u(it,i)   u_anal(it,i)      err(it,i)
 0.00 -30.000       0.000000       0.000000       0.000000
 0.00 -27.500       0.000000       0.000000       0.000000
 0.00 -25.000       0.000000       0.000000       0.000000
 0.00 -22.500       0.000000       0.000000       0.000000
 0.00 -20.000       0.000000       0.000000       0.000000
 0.00 -17.500       0.000000       0.000000       0.000000
 0.00 -15.000       0.000001       0.000001       0.000000
 0.00 -12.500       0.000007       0.000007       0.000000
 0.00 -10.000       0.000091       0.000091       0.000000
 0.00  -7.500       0.001105       0.001105       0.000000
 0.00  -5.000       0.013296       0.013296       0.000000
 0.00  -2.500       0.140207       0.140207       0.000000
 0.00   0.000       0.500000       0.500000       0.000000
 0.00   2.500       0.140207       0.140207       0.000000
 0.00   5.000       0.013296       0.013296       0.000000
 0.00   7.500       0.001105       0.001105       0.000000
 0.00  10.000       0.000091       0.000091       0.000000
 0.00  12.500       0.000007       0.000007       0.000000
 0.00  15.000       0.000001       0.000001       0.000000
 0.00  17.500       0.000000       0.000000       0.000000
 0.00  20.000       0.000000       0.000000       0.000000
        .                                            .
        .                                            .
        .                                            .
```

```
 0.00  60.000       0.000000       0.000000       0.000000
 0.00  62.500       0.000000       0.000000       0.000000
 0.00  65.000       0.000000       0.000000       0.000000
 0.00  67.500       0.000000       0.000000       0.000000
 0.00  70.000       0.000000       0.000000       0.000000

Invariants at t =  0.00
   I1 =     2.0000   Mass conservation
   I2 =     0.3333   Energy conservation
   I3 =     0.4005   Whitham invariant

    t       x        u(it,i)   u_anal(it,i)      err(it,i)
10.00 -30.000       0.000000       0.000000       0.000000
10.00 -27.500      -0.000417       0.000000      -0.000417
10.00 -25.000       0.000103       0.000000       0.000103
10.00 -22.500       0.000193       0.000000       0.000193
10.00 -20.000      -0.000281       0.000000      -0.000281
         .                                          .
         .                                          .
         .                                          .
10.00   0.000      -0.000275       0.000091      -0.000366
10.00   2.500       0.001032       0.001105      -0.000073
10.00   5.000       0.013595       0.013296       0.000299
10.00   7.500       0.141836       0.140207       0.001629
10.00  10.000       0.500778       0.500000       0.000778
10.00  12.500       0.138160       0.140207      -0.002047
10.00  15.000       0.013043       0.013296      -0.000253
10.00  17.500       0.000580       0.001105      -0.000525
10.00  20.000       0.000129       0.000091       0.000038
10.00  22.500       0.000016       0.000007       0.000008
10.00  25.000       0.000007       0.000001       0.000007
10.00  27.500      -0.000331       0.000000      -0.000331
10.00  30.000       0.000172       0.000000       0.000172
         .                                          .
         .                                          .
         .                                          .
10.00  60.000       0.000009       0.000000       0.000009
10.00  62.500      -0.000126       0.000000      -0.000126
10.00  65.000       0.000305       0.000000       0.000305
10.00  67.500       0.000004       0.000000       0.000004
10.00  70.000       0.000000       0.000000      -0.000000

Invariants at t = 10.00
   I1 =     2.0001   Mass conservation
   I2 =     0.3333   Energy conservation
   I3 =     0.4005   Whitham invariant
```

(*continued*)

Table 7.1 (*continued*)

```
    t       x        u(it,i)   u_anal(it,i)      err(it,i)
20.00 -30.000       0.000000       0.000000       0.000000
20.00 -27.500      -0.000151       0.000000      -0.000151
20.00 -25.000       0.000166       0.000000       0.000166
20.00 -22.500      -0.000617       0.000000      -0.000617
20.00 -20.000      -0.000194       0.000000      -0.000194
         .                                          .
         .                                          .
         .                                          .
20.00  10.000       0.000188       0.000091       0.000098
20.00  12.500       0.000271       0.001105      -0.000834
20.00  15.000       0.013899       0.013296       0.000603
20.00  17.500       0.143332       0.140207       0.003124
20.00  20.000       0.501271       0.500000       0.001271
20.00  22.500       0.137520       0.140207      -0.002688
20.00  25.000       0.013101       0.013296      -0.000195
20.00  27.500       0.002060       0.001105       0.000955
20.00  30.000       0.000285       0.000091       0.000194
         .                                          .
         .                                          .
         .                                          .
20.00  60.000      -0.000102       0.000000      -0.000102
20.00  62.500      -0.000177       0.000000      -0.000177
20.00  65.000      -0.000219       0.000000      -0.000219
20.00  67.500       0.000230       0.000000       0.000230
20.00  70.000       0.000000       0.000000      -0.000000

Invariants at t = 20.00
   I1 =     2.0000   Mass conservation
   I2 =     0.3333   Energy conservation
   I3 =     0.4002   Whitham invariant

    t       x        u(it,i)   u_anal(it,i)      err(it,i)
30.00 -30.000       0.000000       0.000000       0.000000
30.00 -27.500      -0.000736       0.000000      -0.000736
30.00 -25.000       0.000314       0.000000       0.000314
30.00 -22.500      -0.000321       0.000000      -0.000321
30.00 -20.000       0.000509       0.000000       0.000509
         .                                          .
         .                                          .
         .                                          .
30.00  20.000       0.000002       0.000091      -0.000089
30.00  22.500       0.000797       0.001105      -0.000308
30.00  25.000       0.013761       0.013296       0.000464
30.00  27.500       0.146476       0.140207       0.006269
30.00  30.000       0.499348       0.500000      -0.000652
30.00  32.500       0.134876       0.140207      -0.005331
```

```
30.00  35.000       0.011331       0.013296      -0.001965
30.00  37.500       0.000444       0.001105      -0.000661
30.00  40.000      -0.000318       0.000091      -0.000409
         .                                          .
         .                                          .
         .                                          .
30.00  60.000      -0.000789       0.000000      -0.000789
30.00  62.500       0.000678       0.000000       0.000678
30.00  65.000       0.000492       0.000000       0.000492
30.00  67.500      -0.001515       0.000000      -0.001515
30.00  70.000       0.000000       0.000000      -0.000000

Invariants at t = 30.00
   I1 =     1.9996   Mass conservation
   I2 =     0.3334   Energy conservation
   I3 =     0.4005   Whitham invariant

   ncall = 5076
```

We can note the following points about this output:

1. The soliton moves left to right in x, as indicated by the movement of the peak. Refer to Table 7.2 for a summary of the peak movement at `t=0,10,20,30` corresponding to `x=0,10,20,30`. In this case, the maximum error in the peak amplitude is `0.001271`, which is 0.25% of the exact value of 0.5.
2. A similar summary of the first invariant (with an exact value of `I1=2`) is given in Table 7.3.

Table 7.2. Partial output from `pde_1_main` and `pde_1` (`ncase=1`), illustrating the movement of the single-soliton peak with velocity $c = 1$

t	x	u(it,i)	u_anal(it,i)	err(it,i)
0.00	0.000	0.500000	0.500000	0.000000
10.00	10.000	0.500778	0.500000	0.000778
20.00	20.000	0.501271	0.500000	0.001271
30.00	30.000	0.499348	0.500000	−0.000652

Table 7.3. Partial output from `pde_1_main` and `pde_1` (`ncase=1`), illustrating the invariant `I1` of Eq. (7.4a)

```
( t = 0) I1 =     2.0000

(t = 10) I1 =     2.0001

(t = 20) I1 =     2.0000

(t = 30) I1 =     1.9996
```

The output of Table 7.3 illustrates two points: (a) the accuracy of the MOL numerical solution for `I1` of Eq. (7.4a), and (7.4b) the accuracy of the Simpson's rule numerical quadrature in routine `simp`. Similar conclusions follow for the invariants `I2` and `I3` of Eqs. (7.4b) and (7.4c).

The properties of the soliton can be visualized from the graphical output of `pde_1` (see Figure 7.1).

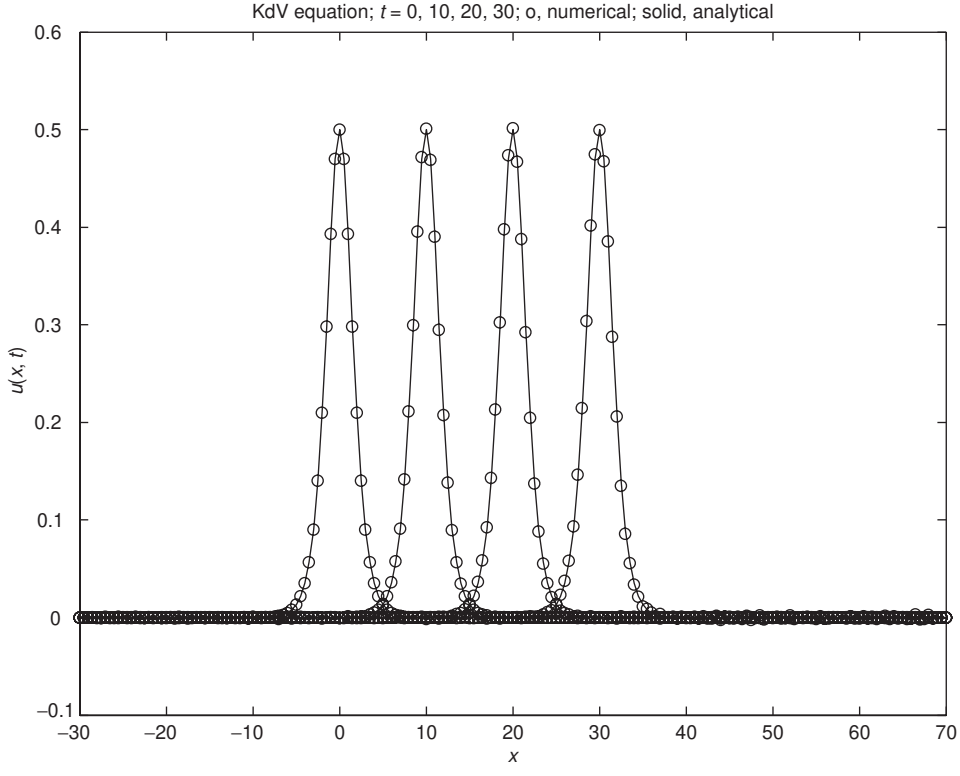

Figure 7.1. Output of main program `pde_1_main` for `ncase=1`

The solution starts with the IC pulse of Eq. (7.3) centered at $x = 0$. The movement of the soliton left to right is evident, as well as the close agreement between the numerical MOL solution and the analytical solution of Eq. (7.2). Note how the soliton retains its shape (a consequence of the traveling wave solution of Eq. (7.2)).

We now consider the subordinate routines called by the main program `pde_1_main` of Listing 7.1, `inital_1, ua, simp` and `pde_1`. `inital_1` is listed first (see Listing 7.2).

```
  function u0=inital_1(t0)
%
% Function inital_1 sets the initial condition for the
% KdV equation
%
  global  ncase     c    c1    c2    xl    xu     x     n
%
% Spatial domain and initial condition
  xl=-30.0;
  xu= 70.0;
  dx=(xu-xl)/(n-1);
%
% Case 1 - single pulse
  if(ncase==1)
    for i=1:n
      x(i)=-30.0+(i-1)*dx;
      u0(i)=ua(x(i),0.0);
    end
  end
%
% Case 2 - two pulses
  if(ncase==2)
    for i=1:n
      x(i)=-30.0+(i-1)*dx;
      expm=exp(-1.0/2.0*sqrt(c1)*(x(i)+15.0));
      expp=exp( 1.0/2.0*sqrt(c1)*(x(i)+15.0));
      pulse1=(1.0/2.0)*c1*(2.0/(expp+expm))^2;
      expm=exp(-1.0/2.0*sqrt(c2)*(x(i)-15.0));
      expp=exp( 1.0/2.0*sqrt(c2)*(x(i)-15.0));
      pulse2=(1.0/2.0)*c2*(2.0/(expp+expm))^2;
      u0(i)=pulse1+pulse2;
    end
  end
```

Listing 7.2. Initialization routine `inital_1`

We can note the following details about `inital_1`:

1. After the routine is defined and some parameters are defined as global, a spatial grid is defined for $-30 \leq x \leq 70$ (recall that the number of grid points, n, is set in main program `pde_1_main` as a global parameter).

```
  function u0=inital_1(t0)
%
% Function inital_1 sets the initial condition for the
% KdV equation
%
  global  ncase     c    c1    c2    xl    xu     x     n
%
% Spatial domain and initial condition
  xl=-30.0;
  xu= 70.0;
  dx=(xu-xl)/(n-1);
```

   This spatial interval is sufficiently long that the solution of Eq. (7.1) does not depart from the IC of Eq. (7.3) near $x = -30, 70$ so that this interval is effectively infinite. The consequence of this is that BCs for Eq. (7.1) are not required.
2. The IC for `ncase=1` is then defined by a call to the analytical solution of Eq. (7.2) in `ua` with $t = 0$.

```
%
% Case 1 - single pulse
  if(ncase==1)
    for i=1:n
      x(i)=-30.0+(i-1)*dx;
      u0(i)=ua(x(i),0.0);
    end
  end
```

3. Finally, the two-soliton IC `ncase=2` is defined.

```
%
% Case 2 - two pulses
  if(ncase==2)
    for i=1:n
      x(i)=-30.0+(i-1)*dx;
```

```
      expm=exp(-1.0/2.0*sqrt(c1)*(x(i)+15.0));
      expp=exp( 1.0/2.0*sqrt(c1)*(x(i)+15.0));
      pulse1=(1.0/2.0)*c1*(2.0/(expp+expm))^2;
      expm=exp(-1.0/2.0*sqrt(c2)*(x(i)-15.0));
      expp=exp( 1.0/2.0*sqrt(c2)*(x(i)-15.0));
      pulse2=(1.0/2.0)*c2*(2.0/(expp+expm))^2;
      u0(i)=pulse1+pulse2;
    end
  end
```

Note the use of the velocity `c1=2` for the first soliton (`pulse1`) centered at $x = -15$ and the velocity `c2=0.5` for the second soliton (`pulse2`) centered at $x = 15$; these velocities are set as global variables in the main program `pde_1_main`.

The analytical solution of Eq. (7.2) is programmed in `ua` (see Listing 7.3).

```
  function uanal=ua(x,t)
%
% Function uanal computes the exact solution of the
% KdV equation for comparison with the numerical solution.
%
  global   c
%
% Analytical solution
  expm=exp(-1.0/2.0*sqrt(c)*(x-c*t));
  expp=exp( 1.0/2.0*sqrt(c)*(x-c*t));
  uanal=(1.0/2.0)*c*(2.0/(expp+expm))^2;
```

Listing 7.3. Analytical solution routine `ua` for Eq. (7.2)

The code in `ua` is essentially self-explanatory when compared with Eq. (7.2) (recall $\text{sech}(x) = 1/\cosh(x) = 2/(e^x + e^{-x})$ and note the argument for the traveling wave solution, $x - ct$, with $t = 0$ when `ua` is called from `inital_1` for IC (7.3)).

The routine for calculating the integrals of Eqs. (7.4a)–(7.4c) by Simpson's rule, `simp`, is given in Listing 7.4.

```
  function uint=simp(xl,xu,n,u)
%
% Function simp computes three integral invariants by Simpson's
% rule
%
```

```
  for int=1:3
    h=(xu-xl)/(n-1);
%
%   Conservation of mass
    if(int==1)
       uint(1)=u(1)-u(n);
       for i=3:2:n
         uint(1)=uint(1)+4.0*u(i-1)+2.0*u(i);
       end
       uint(1)=h/3.0*uint(1);
    end
%
%   Conservation of energy
    if(int==2)
       uint(2)=u(1)^2-u(n)^2;
       for i=3:2:n
         uint(2)=uint(2)+4.0*u(i-1)^2+2.0*u(i)^2;
       end
       uint(2)=(1.0/2.0)*h/3.0*uint(2);
    end
%
%   Whitham conservation
    if(int==3)
       ux=dss004(xl,xu,n,u);
       uint(3)=2.0*u(1)^3-ux(1)^2-(2.0*u(n)^3-ux(n)^2);
       for i=3:2:n
         uint(3)=uint(3)+4.0*(2.0*u(i-1)^3-ux(i-1)^2)...
                        +2.0*(2.0*u(i)^3  -ux(i)^2);
       end
       uint(3)=h/3.0*uint(3);
    end
  end
```

Listing 7.4. Numerical quadrature routine `simp` applied to Eqs. (7.4a)–(7.4c)

`simp` has three parts corresponding to the integrals `I1,I2,I3` of Eqs. (7.4a)–(7.4c).

1. The coding for `I1` is

```
  function uint=simp(xl,xu,n,u)
%
% Function simp computes three integral invariants by Simpson's
% rule
%
```

```
  for int=1:3
    h=(xu-xl)/(n-1);
%
%   Conservation of mass
    if(int==1)
       uint(1)=u(1)-u(n);
       for i=3:2:n
         uint(1)=uint(1)+4.0*u(i-1)+2.0*u(i);
       end
       uint(1)=h/3.0*uint(1);
    end
```

After defining the function, the integration interval `h=(xu-xl)/(n-1)` is computed, where `xl=-30 and xu=70` are the lower and upper limits of the integral (set in `inital_1`). The `for` loop for `I1` is an implementation of the weighted sum for Simpson's rule applied to the function $u(x, t)$ (the integrand in Eq. (7.4a)).

$$\int_{-\infty}^{\infty} u(x,t)dx \approx \frac{h}{3}\left[u_1 + \sum_{i=2}^{n-2}(4u_i + 2u_{i+1}) + u(n)\right]$$

2. Similarly, the coding for `I2` of Eq. (7.4b) is

```
%
%   Conservation of energy
    if(int==2)
       uint(2)=u(1)^2-u(n)^2;
       for i=3:2:n
         uint(2)=uint(2)+4.0*u(i-1)^2+2.0*u(i)^2;
       end
       uint(2)=(1.0/2.0)*h/3.0*uint(2);
    end
```

The only difference is the use of the integrand $u(x, t)^2$ according to Eq. (7.4b).

3. Finally, the coding for `I3` of Eq. (7.4c) is

```
%
%   Whitham conservation
    if(int==3)
       ux=dss004(xl,xu,n,u);
       uint(3)=2.0*u(1)^3-ux(1)^2-(2.0*u(n)^3-ux(n)^2);
```

```
      for i=3:2:n
        uint(3)=uint(3)+4.0*(2.0*u(i-1)^3-ux(i-1)^2)...
                        +2.0*(2.0*u(i)^3  -ux(i)^2);
      end
      uint(3)=h/3.0*uint(3);
    end
  end
```

The integrand is $2u(x,t)^3 - u_x(x,t)^2$ according to Eq. (7.4c). The derivative $u_x(x,t)$ is computed by a call to the differentiation routine `dss004`.

The MOL ODE routine, `pde_1`, is given in Listing 7.5.

```
  function ut=pde_1(t,u)
%
% Function pde_1 computes the t derivative vector for the
% KdV equation
%
  global  n xl xu ncall
%
% Calculate ux
  ux=dss004(xl,xu,n,u);
%
% Calculate uxxx
  uxxx=uxxx7c(xl,xu,n,u);
%
% PDE
  for i=1:n
    ut(i)=-uxxx(i)-6.0*u(i)*ux(i);
  end
  ut=ut';
%
% Increment calls to pde_1
  ncall=ncall+1;
```

Listing 7.5. MOL ODE routine `pde_1` called by main program `pde_1_main`

We can note the following points about `pde_1`:

1. After the definition of the function and the global declaration of some parameters and variables, the first derivative u_x in Eq. (7.1) is computed by a call to `dss004`.

```
  function ut=pde_1(t,u)
%
% Function pde_1 computes the t derivative vector for the
% KdV equation
%
  global  n xl xu ncall
%
% Calculate ux
  ux=dss004(xl,xu,n,u);
```

2. The third derivative u_{xxx} in Eq. (7.1) is then computed by a call to `uxxx7c`, which has a seven-point finite-difference (FD) approximation for u_{xxx}.

```
%
% Calculate uxxx
  uxxx=uxxx7c(xl,xu,n,u);
```

An alternative to the use of `uxxx7c` would be to apply three successive differentiations to `u` using `dss004`; that is, $u \rightarrow u_x \rightarrow u_{xx} \rightarrow u_{xxx}$. This *stagewise differentiation* has been used to good effect in the MOL analysis of higher-order PDEs. However, we chose here to use `uxxx7c` to illustrate the rather specialized programming for a higher-order derivative. `uxxx7c` is listed in Appendix A at the end of this chapter.

3. Finally, the programming of Eq. (7.1) in a `for` loop over the spatial grid of n points is straightforward.

```
%
% PDE
  for i=1:n
    ut(i)=-uxxx(i)-6.0*u(i)*ux(i);
  end
  ut=ut';
%
% Increment calls to pde_1
  ncall=ncall+1;
```

Note in particular the close resemblance of the coding to Eq. (7.1), which is a major advantage of the MOL approach to the solution of PDE systems.

`ncall` is incremented and displayed at the end of the solution (in `pde_1_main`) to give an indication of the computational effort to produce the solution (the number of calls to `pde_1`).

The `for` loop for the PDE in `pde_1` is an example of `procedural programming` that is common to most languages used for scientific computation, for example, Fortran, C, C++, Java. In Matlab, the subscripting in the `for` loop can be replaced by vector operations that are specific to Matlab. The advantages are basically twofold: The code is simplified and more efficient. To illustrate this approach, the preceding procedural code

```
%
% PDE
  for i=1:n
    ut(i)=-uxxx(i)-6.0*u(i)*ux(i);
  end
  ut=ut';
```

can be replaced with the vectorized code

```
%
% PDE
  ut=-uxxx'-6.0*u.*ux';
```

We note the following points:

1. The subscripting in the `for` loop (in terms of `i`) is eliminated and all of the variables are now vectors.
2. The nonlinear term uu_x in Eq. (7.1) is programmed using the vector multiplication `.*`, where "`.`" specifies an *element-by-element* operation, in this case multiplication.
3. The RHS involves the formation of two *column vectors*, `-6.0*u.*ux'` and `-uxxx'`, which are then subtracted (`-uxxx'-6.0*u.*ux'`) to form the column vector `ut` (so that a transpose of `ut` is not required; i.e., `u` is already a column vector).
4. The differentiation routines `dss004` and `uxxx7c` return `row vectors`, which then must be transposed to column vectors.

Of course, this code could be simplified further if `dss004` and `uxxx7c` are programmed to return column vectors (so that the transposes are not required).

As a consequence of using the sparse version of `ode15s` for the integration of the n ODEs, a *map of the Jacobian matrix* of the ODE system is produced (by `ode15s`) (see Figure 7.2). We will not discuss the concept of the Jacobian matrix of an ODE system other than to point out that it displays the relationship between the derivative of the ith ODE, dy_i/dt, and the dependent variables on which this derivative depends; that is,

$$dy_i/dt = f_i(y_1, y_2, \ldots, y_{n-1}, y_n) \tag{7.5}$$

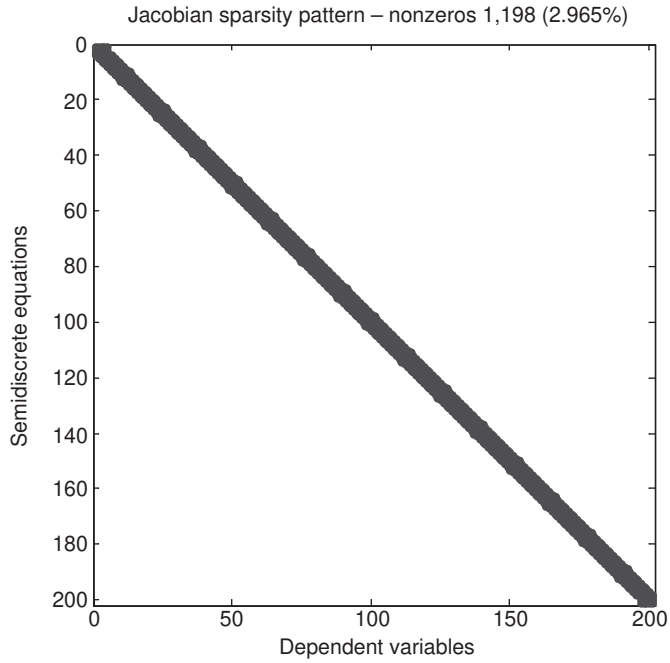

Figure 7.2. Jacobian matrix map for the system of n ODEs

For example, if dy_4/dt is a function of $y_1, y_2, y_3, y_4, y_5, y_6, y_7$ (e.g., seven values of y as would be the case for the seven-point FDs used in `uxxx7c`), then the fourth row of the map would have *seven* entries. This is the case of the map of Figure 7.2 because of the use of seven-point FDs in `uxxx7c`. Thus, the ODE system mapped in Figure 7.2 is said to be *banded* with a *bandwidth* of 7.

Note that only 2.965% of the $n \times n$ elements of the Jacobian map of Figure 7.2 are nonzero. This is not uncommon; that is, most of the elements are zero. This condition is the reason for the use of sparse matrix integrators since they in general will process only the *nonzero elements* and will not expend computer time processing the many zero elements. This saving in computer time can be very substantial, for example, *orders of magnitude*, so that the additional logic of the sparse matrix integrator (to detect the nonzero elements of the Jacobian matrix and then perform the numerical integration using only these nonzero elements) is well worth the additional complexity of the coding for the sparse matrix.

In order to produce a Jacobian map such as Figure 7.2, the sparse matrix option of `ode15s` requires a user-supplied routine `jpattern_num`. The name of this routine indicates that the $n \times n$ elements of the Jacobian matrix (n is the number of ODEs) are computed numerically (by FDs); the elements of the Jacobian matrix are actually the partial derivatives $((\partial dy_i/dt)/\partial y_j) = \partial f_i/\partial y_j$, where f_i is the derivative function from Eq. (7.5). An alternative approach to calculating the Jacobian matrix would be to analytically derive the partial derivatives $\partial f_i/\partial y_j$. However, this is generally impractical because of the number of such partial derivatives; for example, for $n = 1{,}000$, the Jacobian matrix has $1{,}000^2 = 1{,}000{,}000$ elements. In other words, numerical calculation of the Jacobian matrix is the only feasible approach.

The routine jpattern_num used in the present example, which produced the map of Figure 7.2, is listed in Appendix B at the end of this chapter. We chose to not go into the details of this because of space limitations. However, this routine is quite general and can therefore be applied to other ODE/PDE systems. jpattern_num has calls to several Matlab utilities. We will just mention the changes that might be required for other problems:

- The elements of the Jacobian matrix are computed by FDs. The FDs require a base point around which the numerical derivative is computed. The statement in jpattern_num that sets this base point is

```
ybase(i)=0.5;
```

  The value (0.5 in this case) need only be representative of the range of values of the ODE dependent variables. Since the solution in Figure 7.1 indicates a range of values of 0 to 1 in the solution, we chose 0.5. But again, a precise value is not required (but if ode15s fails, some experimentation with this value may be required).
- In order to calculate the partial derivatives in the Jacobian matrix by FD approximations, $((\partial dy_i/dt)/\partial y_j)$, the derivatives at the base point, dy_i/dt, are also required.

```
ytbase=pde_1(tbase,ybase);
```

  Note that the routine that calculates the derivatives dy_i/dt in Eq. (7.5) is called, in this case, pde_1 of Listing 7.5.
- The elements of the Jacobian matrix are evaluated numerically by a call to the Matlab routine numjac. Since these elements are the partial derivatives of the derivative functions $dy_i/dt = f_i$ in Eq. (7.5), the name of the routine that evaluates these functions must be provided as the first argument of numjac, which is pde_1 of Listing 7.5.

```
[Jac,fac]=numjac(@pde_1,tbase,ybase,ytbase,thresh,fac,
                 vectorized);
```

  Once the Jacobian matrix has been defined by this call to numjac, the map of the Jacobian can be plotted by the code that follows the call to numjac (see the listing in Appendix B).

We can add a few additional points about sparse matrix integration:

1. If PDE systems produced only banded Jacobians (with all of the nonzero elements concentrated around the main diagonal), then a *banded integrator* would be even more efficient than the *sparse integrator* since the banded integrator would know in advance where the nonzero elements occur (all are in the band) and it would therefore not have to search for the nonzero elements, and follow the resulting logic to use these nonzero elements, all of which add complexity to a sparse integrator.
2. However, the banded system of Figure 7.2 is not typical of a MOL PDE approximation. A more typical situation is to have some of the nonzero elements located along the main diagonal (as in a banded system), but to have others located outside the band (so that a banded integrator would miss using these *out-of-band elements*). Then the sparse integrator would be particularly effective since it would locate the out-of-band elements, the so-called *outliers*, and use them as well in the numerical integration, but it would not consume time processing the many zero elements. Thus, this combination of elements in a band along the main diagonal, plus outliers, leads to the efficiency of sparse matrix integration. Such a situation typically results from (a) PDE systems and (b) 2D and 3D PDEs (rather than the single, 1D problem considered here, Eq. (7.1)).
3. The processing of the ODE Jacobian matrix is a requirement for *stiff* (or *implicit*) integrators. Generally, this additional effort of processing the Jacobian matrix is well worth the effort if the ODEs are stiff, as they frequently are for PDE problems. However, if the ODEs are not stiff, then a *nonstiff* (or *explicit*) integrator should be used since they *do not require processing of the Jacobian matix*. This point is illustrated by the current PDE problem (Eq. (7.1)). If `ode45` is called in `pde_1_main` (`mf=1`), a nonstiff integrator, the number of calls to `pde_1` increases substantially relative to `ode15s`, a stiff integrator (as reflected in the larger final value of `ncall`). For borderline cases between stiff and nonstiff ODEs, some experimentation is generally required to determine which type of integrator is more efficient (or in other words, each type of integrator can be useful, depending on the stiffness characteristics of the problem; a common fallacy is to use a stiff integrator in all cases).
4. Returning to the idea of stagewise differentiation, each time a derivative is calculated, for example, u_{xx} from u_x, the bandwidth of the MOL ODEs increases. Thus, for example, if a five-point FD approximation is used to compute a derivative, the final bandwidth after three such numerical differentiations in calculating u_{xxx} would be substantially greater than the bandwidth of seven from the use of `uxx7c`. Since an increased bandwidth means more computational effort in processing the ODE Jacobian matrix, stagewise differentiation might not be as efficient as a direct calculation of the higher-order derivative (as in `uxxx7c`). There is also the matter of the accuracy of stagewise versus direct numerical differentiation, but we will not consider this further.

We now consider the execution of the previous routines for the case `ncase=2` (two solitons) to demonstrate a second quite remarkable property of solitons (in addition to maintaining their shape exactly with increasing t). All that is required to make the additional run described next is to change `ncase` to 2 in `pde_1_main`.

The numerical output for this case will not be considered in detail for two reasons:

1. The output is more complicated than for `ncase=1` (as listed in Table 7.1) and is therefore not easily examined in tabular form.
2. An analytical solution for the two-soliton case is not presented, so the errors in the numerical solution are not available.

We do, however, list the invariants at $t = 0, 10, 20, 30$:

```
Invariants at t =  0.00
   I1 =     4.2426   Mass conservation
   I2 =     1.0607   Energy conservation
   I3 =     2.3358   Whitham invariant

Invariants at t = 10.00
   I1 =     4.2424   Mass conservation
   I2 =     1.0607   Energy conservation
   I3 =     2.3365   Whitham invariant

Invariants at t = 20.00
   I1 =     4.2423   Mass conservation
   I2 =     1.0606   Energy conservation
   I3 =     2.3321   Whitham invariant

Invariants at t = 30.00
   I1 =     4.2433   Mass conservation
   I2 =     1.0630   Energy conservation
   I3 =     2.2826   Whitham invariant

 ncall = 14329
```

Note that the invariants have different values than for the one-soliton case. They do, however, remain essentially constant (since `I1,I2,I3` of Eqs. (7.4a)–(7.4c) are properties of Eq. (7.1), and are not dependent on the particular initial condition used, e.g., one or two solitons).

We now present the plotted output from `pde_1_main` (see Figure 7.3). We note the following points about Figure 7.3:

1. At $t = 0$, the higher soliton (with a peak value of 1 from `inital_1`) is to the left of the lower soliton (with a peak value of 0.25). This relative positioning is important since the higher soliton will travel left to right faster than the lower soliton.
2. At $t = 10$, the higher soliton has moved closer to the lower soliton.
3. At $t = 20$, the two solitons have merged.

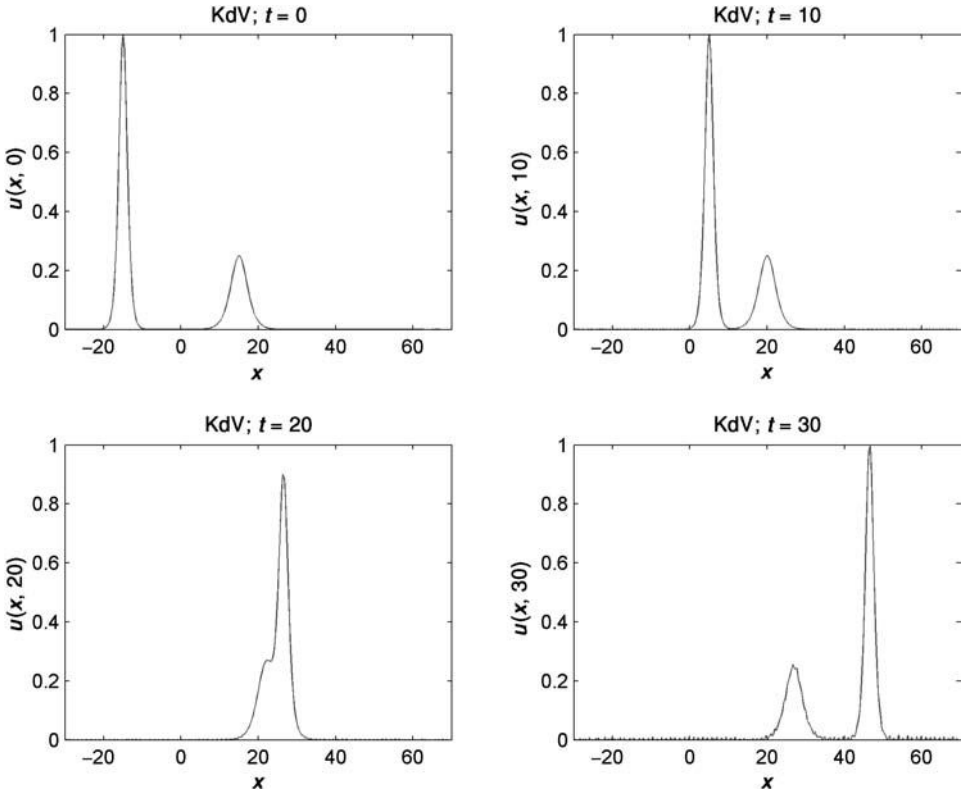

Figure 7.3. Output of main program `pde_1_main` for `ncase=2`

4. At $t = 30$, the solitons have again separated with the higher soliton now to the right of the lower soliton. Also, the two solitons have retained their shapes from $t = 0$.

Finally, to further elucidate the numerical solution for `ncase=2`, we also include a 3D plot produced with the following code:

```
% 3D Plots
 figure()
 surfl(x,t,u, 'light'); shading interp
 axis tight
 title('Korteweg-de Vries (KdV) equation');
 set(get(gca,'XLabel'),'String','space, x')
 set(get(gca,'YLabel'),'String','time, t')
 set(get(gca,'ZLabel'),'String','u(x,t)')
 set(gca,'XLim',[xl xu],'YLim',[t0 tf],'ZLim', [0 1]);
 view(-10,70);
 colormap('cool');
 print -dpng -r300 fig4.png;
```

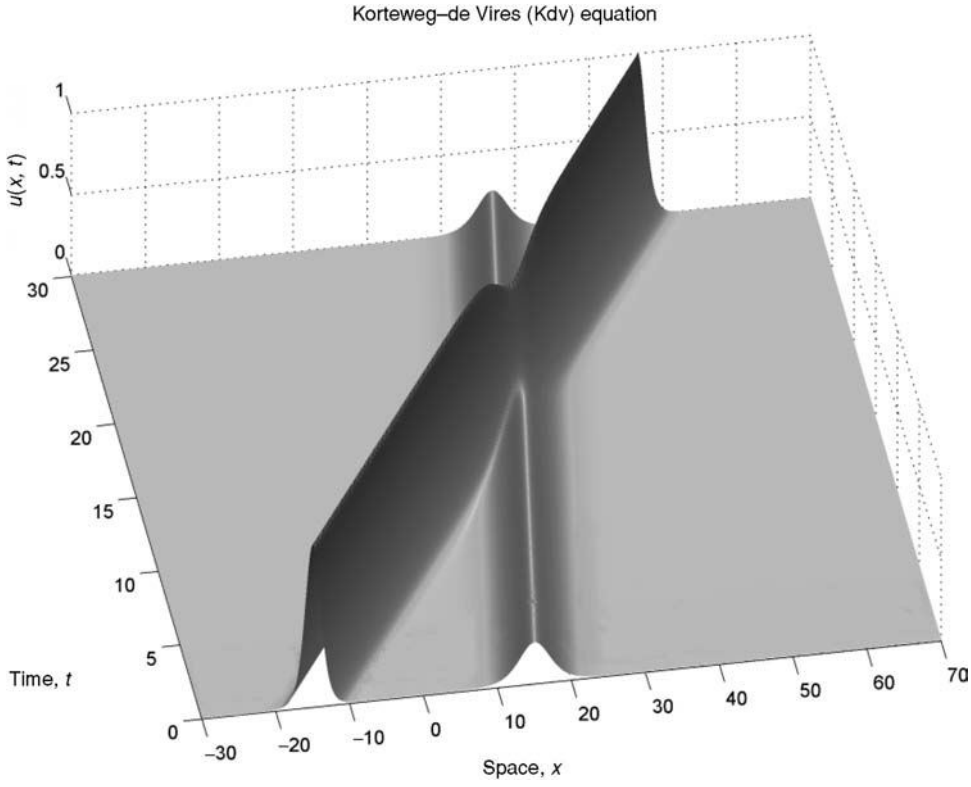

Figure 7.4. Three-dimensional output for main program `pde_1_main` with `ncase=2, nout=201, n=1001`

The resulting 3D plot is shown in Figure 7.4. Additional enhanced plotting of the solution of Eq. (7.1) can be accomplished by animation (a movie). The details for producing an animation are given in Appendix 6.

We conclude this chapter with this demonstration of the quite remarkable property of Eq. (7.1), the merging and subsequent emergence of the two solitons.

APPENDIX A

A.1. FD Routine uxxx7c

```
  function uxxx=uxxx7c(xl,xu,n,u)
%
% Function uxxx7c computes the derivative uxxx in the
% KdV equation
%
% Spatial increment
  dx=(xu-xl)/(n-1);
```

```
  r8dx3=1.0/(8.0*(dx^3));
%
% uxxx
  for i=1:n
%
%   At the left end, uxxx = 0
    if(i<4)uxxx(i)=0.0;
%
%   At the right end, uxxx = 0
    elseif(i>(n-3))uxxx(i)=0.0;
%
%   Interior points
    else
    uxxx(i)=r8dx3*...
       (   1.0*u(i-3)...
          -8.0*u(i-2)...
         +13.0*u(i-1)...
          +0.0*u(i  )...
         -13.0*u(i+1)...
          +8.0*u(i+2)...
          -1.0*u(i+3));
    end
  end
```

Note that the derivative u_{xxx} (= `uxxx(i)`) is calculated as a weighted sum of the seven values

```
u(i-3),u(i-2),u(i-1),u(i),u(i+1),u(i+2),u(i+3)
```

These values are centered around the point `i`, so this is a *seven-point, centered FD approximation*. The weighting coefficients are computed from standard formulas, but we will not consider the details here. At the ends of the interval in x (`i=1,2,3,n-2,n-1,n`), the derivative u_{xxx} is set to zero since we have assumed that these boundaries do not affect the solution (see Figures 7.1 and 7.3 to support this assumption).

APPENDIX B

B.1. Jacobian Matrix Routine `jpattern_num`

```
  function S=jpattern_num
%
  global n
%
% Sparsity pattern of the Jacobian matrix based on a
% numerical evaluation
```

```
%
% Set independent, dependent variables for the calculation
% of the sparsity pattern
  tbase=0;
  for i=1:n
    ybase(i)=0.5;
  end
  ybase=ybase';
%
% Compute the corresponding derivative vector
  ytbase=pde_1(tbase,ybase);
  fac=[];
  thresh=1e-16;
  vectorized='on';
  [Jac,fac]=numjac(@pde_1,tbase,ybase,ytbase,thresh,fac, ...
                   vectorized);
%
% Replace nonzero elements by "1" (so as to create a "0-1" map
% of the Jacobian matrix)
  S=spones(sparse(Jac));
%
% Plot the map
  figure
  spy(S);
  xlabel('dependent variables');
  ylabel('semi-discrete equations');
%
% Compute the percentage of non-zero elements
  [njac,mjac]=size(S);
  ntotjac=njac*mjac;
  non_zero=nnz(S);
  non_zero_percent=non_zero/ntotjac*100;
  stat=sprintf('Jacobian sparsity pattern - nonzeros %d
        (%.3f%%)', non_zero,non_zero_percent);
  title(stat);
```

APPENDIX C

C.1. Some Background to the KdV Equation

Equation (7.1) is the conventional, nondimensional version of the following equation originally derived by Korteweg and de Vries for a *moving (Lagrangian)* frame of reference [2, 4]:

$$\frac{\partial \eta}{\partial \tau} = \frac{3}{2}\sqrt{\frac{g}{h_o}}\frac{\partial}{\partial \chi}\left[\frac{1}{2}\eta^2 + \frac{2}{3}\alpha\eta + \frac{1}{3}\sigma\frac{\partial^2 \eta}{\partial \chi^2}\right]$$

It describes small-amplitude, shallow-water waves in a channel where $\sigma = h_o^3/3 - \mathcal{T}h_o/(\rho g)$ and symbols have the following meaning:

g	gravitational acceleration (m/s^2)
h_o	nominal water depth (m)
T	capillary surface tension of fluid (N/m)
α	small arbitrary constant related to the uniform motion of the liquid (dimensionless)
η	wave height (m)
ρ	fluid density (kg/m^3)
τ	time (s)
χ	distance (m)

After rescaling and translating the dependent and independent variables to eliminate the physical constants using the transformations [5],

$$u = -\frac{1}{2}\eta - \frac{1}{3}\alpha; \qquad x = -\frac{\chi}{\sqrt{\sigma}}; \qquad t = \frac{1}{2}\sqrt{\frac{g}{h_o\sigma}}\tau$$

we arrive at the "canonical" form of the KdV equation, $u_t - 6uu_x + u_{\{xxx\}} = 0$, and this can be changed to the "alternative" form of Eq. (7.1) $u_t + 6uu_x + u_{\{xxx\}} = 0$ through the change of variable $u = -u$.

The KdV equation was found to have solitary wave solutions [6], which confirmed John Scott-Russell's account of the solitary wave phenomena [7] discovered during his experimental investigations into water flow in channels to determine the most efficient design for canal boats [4]. John Scott-Russell also described in poetic terms his first encounter with the solitary wave phenomena; thus,

> I was observing the motion of a boat which was rapidly drawn along a narrow channel by a pair of horses, when the boat suddenly stopped – not so the mass of water in the channel which it had put in motion; it accumulated round the prow of the vessel in a state of violent agitation, then suddenly leaving it behind, rolled forward with great velocity, assuming the form of a large solitary elevation, a rounded, smooth and well-defined heap of water, which continued its course along the channel apparently without change of form or diminution of speed. I followed it on horseback, and overtook it still rolling on at a rate of some eight or nine miles an hour, preserving its original figure some thirty feet long and a foot to a foot and a half in height. Its height gradually diminished, and after a chase of one or two miles I lost it in the windings of the channel. Such, in the month of August 1834, was my first chance interview with that singular and beautiful phenomenon which I have called the Wave of Translation [7].

The term *solitary wave* was also coined by Russell.

REFERENCES

[1] Debnath, L. (1997), Solitons and the Inverse Scattering Transform, *Nonlinear Partial Differential Equations for Scientists and Engineers*, Birkhauser, Boston, Chapter 9, pp. 331–404

[2] Korteweg, D. J. and F. de Vries (1895), On the Change of Form of Long Waves Advancing in a Rectangular Canal, and on a New Type of Long Stationary Waves, *Phil. Mag*. **39**:422–443

[3] Strang, G. (1986), *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Wellesley, MA, pp. 599–602

[4] de Jager, E. M. (2006), On the Origin of the Korteweg–deVries Equation, *arXiv e-print service*; available online at arXiv.org `http://arxiv.org/PS_cache/math/pdf/0602/0602661v1.pdf`

[5] Usman, M. (2007), *Forced Oscillations of the Korteweg–deVries Equation and Their Stability*, Ph.D. Thesis, McMicken College of Arts & Sciences, University of Cincinnati OH

[6] Lamb, H. (1993), *Hydrodynamics,* 6th ed., Cambridge University Press, Cambridge, UK, pp. 422–424

[7] Scott-Russell, J. (1844), Report on Waves, In: *14th Meeting of the British Association for the Advancement of Science*, John Murray, London, pp. 311–391, plates XLVII–LVII