pde_method_of_lines

Solves a system of partial differential equations of the form $u_t = f(x, t, u, u_x, u_{xx})$ using the method of lines. The solution is represented with cubic Hermite polynomials.

Synopsis

#include `<imsl.h>`

void `imsl_f_pde_method_of_lines_mgr` (*int* `task`, *void* `**state`, ..., 0)

void `imsl_f_pde_method_of_lines` (*int* `npdes`, *float* `*t`, *float* `tend`, *int* `nx`, *float* `xbreak[]`, *float* `y[]`, *void* `*state`, *void* `fcn_ut()`, *void* `fcn_bc()`)

The type *double* functions are `imsl_d_pde_method_of_lines_mgr` and `imsl_d_pde_method_of_lines`.

Required Arguments for imsl_f_pde_method_of_lines_mgr

int `task` (Input)
This function must be called with task set to `IMSL_PDE_INITIALIZE` to set up memory and default values prior to solving a problem and with task equal to `IMSL_PDE_RESET` to clean up after it has solved. These values for task are defined in the header file `imsl.h`.

void `**state` (Input/Output)
The current state of the PDE solution is held in a structure pointed to by `state`. It cannot be directly manipulated.

Required Arguments for imsl_f_pde_method_of_lines

int `npdes` (Input)
Number of differential equations.

float `*t` (Input/Output)
Independent variable. On input, *t* supplies the initial time, t_0. On output, *t* is set to the value to which the integration has been updated. Normally, this new value is `tend`.

float `tend` (Input)
Value of *t* = `tend` at which the solution is desired.

int `nx` (Input)
Number of mesh points or lines.

float `xbreak[]` (Input)
Array of length `nx` containing the breakpoints for the cubic Hermite splines used in the

x discretization. The points in `xbreak` must be strictly increasing. The values `xbreak`[0] and `xbreak`[`nx` – 1] are the endpoints of the interval.

float `y[]` (Input/Output)
Array of size `npdes` by `nx` containing the solution. The array `y` contains the solution as *y*[`k`,`i`] = u_k(`x`, `tend`) at *x* = `xbreak`[`i`]. On input, *y* contains the initial values. It must satisfy the boundary conditions. On output, `y` contains the computed solution.

void `*state` (Input/Output)
The current state of the PDE solution is held in a structure pointed to by state. It must be initialized by a call to `imsl_f_pde_method_of_lines_mgr`. It cannot be directly manipulated.

void `fcn_ut`(*int* `npdes`, *float* `x`, *float* `t`, *float* `u[]`, *float* `ux[]`, *float* `uxx[]`, *float* `ut[]`)
User-supplied function to evaluate u_t.

int `npdes` (Input)
Number of equations.

float `x` (Input)
Space variable, *x*.

float `t` (Input)
Time variable, *t*.

float `u[]` (Input)
Array of length `npdes` containing the dependent values, *u*.

float `ux[]` (Input)
Array of length `npdes` containing the first derivatives, u_x.

float `uxx[]` (Input)
Array of length `npdes` containing the second derivative, u_{xx}.

float `ut[]` (Output)
Array of length `npdes` containing the computed derivatives u_t.

void `fcn_bc`(*int* `npdes`, *float* `x`, *float* `t`, *float* `alpha[]`, *float* `beta[]`, *float* `gammap[]`)
User-supplied function to evaluate the boundary conditions. The boundary conditions accepted by `imsl_f_pde_method_of_lines` are

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k$$

Note: Users must supply the values α_k and β_k, which determine the values γ_k. Since γ_k can depend on *t* values of γ_k' also are required.

int `npdes` (Input)
Number of equations.

float `x` (Input)
Space variable, *x*.

float `t` (Input)
Time variable, *t*.

float `alpha[]` (Output)
Array of length `npdes` containing the α_k values.

float `beta[]` (Output)
Array of length `npdes` containing the β_k values.

float `gammap[]` (Output)
Array of length `npdes` containing the derivatives,

$$\frac{d\gamma_k}{dt} = \gamma_k'$$

Synopsis with Optional Arguments

```
#include <imsl.h>
void imsl_f_pde_method_of_lines_mgr (int task, void **state,
       IMSL_TOL, float tol,
       IMSL_HINIT, float hinit,
       IMSL_INITIAL_VALUE_DERIVATIVE, float initial_deriv[],
       IMSL_HTRIAL, float *htrial,
       IMSL_FCN_UT_W_DATA, void fcn_ut (), void *data,
       IMSL_FCN_BC_W_DATA, void fcn_bc (), void *data,
        0)
```

Optional Arguments

`IMSL_TOL`, *float* `tol` (Input)
Differential equation error tolerance. An attempt is made to control the local error in such a way that the global relative error is proportional to `tol`.
Default: `tol` = 100.0`*imsl_f_machine`(4)

`IMSL_HINIT`, *float* `hinit` (Input)
Initial step size in the *t* integration. This value must be nonnegative. If `hinit` is zero, an initial step size of 0.001|`tend` - t_0| will be arbitrarily used. The step will be applied in the direction of integration.
Default: `hinit` = 0.0

`IMSL_INITIAL_VALUE_DERIVATIVE`, *float* `initial_deriv[]` (Input/Output)
Supply the derivative values $u_x(x, t_0)$. This derivative information is input as

$$\texttt{initial\_deriv(k,i)} = \frac{\partial u_k}{\partial x}\left(x, t(0)\right)$$

The array `initial_deriv` contains the derivative values as output:

$$\texttt{initial\_deriv(k,i)} = \frac{\partial u_k}{\partial x}\left(x\texttt{tend}\right) \quad \text{at } x = x[i]$$

Default: Derivatives are computed using cubic spline interpolation

`IMSL_HTRIAL`, *float* `*htrial` (Output)
Return the current trial step size.

`IMSL_FCN_UT_W_DATA`, *void* `fcn_ut`(*int* `npdes`, *float* `x`, *float* `t`, *float* `u[]`, *float* `ux[]`, *float* `uxx[]`, *float* `ut[]`, *void* `*data`), *void* `*data` (Input)
User-supplied function to evaluate u_t, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

`IMSL_FCN_BC_W_DATA`, *void* `fcn_bc`(*int* `npdes`, *float* `x`, *float* `t`, *float* `alpha[]`, *float* `beta[]`, *float* `gammap[]`, *void* `*data`), *void* `*data` (Input)
User-supplied function to evaluate the boundary conditions, which also accepts a pointer to data that is supplied by the user. data is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

Let M = `npdes`, N = `nx` and x_i = `xbreaK(I)`. The routine `imsl_f_pde_method_of_lines` uses the method of lines to solve the partial differential equation system

$$\frac{\partial u_k}{\partial t} = f_k\left(x, t, u_1, \ldots u_M, \frac{\partial u_1}{\partial x}, \ldots \frac{\partial u_M}{\partial x}, \frac{\partial^2 u_1}{\partial x^2}, \ldots \frac{\partial^2 u_M}{\partial x^2}\right)$$

with the initial conditions

$$u_k = u_k(x, t) \qquad \text{at } t = t_0$$

and the boundary conditions

$$\alpha_k u_k + \beta_k \frac{\partial u_k}{\partial x} = \gamma_k \qquad \text{at } x = x_1 \text{ and at } x = x_N$$

for $k = 1, \ldots, M$.

Cubic Hermite polynomials are used in the x variable approximation so that the trial solution is expanded in the series

$$\hat{u}_k(x,t) = \sum_{i=1}^{N} \left(a_{i,k}(t)\phi_i(x) + b_{i,k}(t)\psi_i(x) \right)$$

where $\phi_i(x)$ and $\psi_i(x)$ are the standard basis functions for the cubic Hermite polynomials with the knots $x_1 < x_2 < \ldots < x_N$. These are piecewise cubic polynomials with continuous first derivatives. At the breakpoints, they satisfy

$$\phi_i(x_l) = \delta_{il} \qquad \psi_i(x_l) = 0$$
$$\frac{d\phi_i}{dx}(x_l) = 0 \qquad \frac{d\psi_i}{dx}(x_l) = \delta_{il}$$

According to the collocation method, the coefficients of the approximation are obtained so that the trial solution satisfies the differential equation at the two Gaussian points in each subinterval,

$$p_{2j-1} = x_j + \frac{3-\sqrt{3}}{6}\left(x_{j+1} - x_j\right)$$
$$p_{2j} = x_j + \frac{3-\sqrt{3}}{6}\left(x_{j+1} + x_j\right)$$

for $j = 1, \ldots, N$. The collocation approximation to the differential equation is

$$\frac{da_{i,k}}{dt}\phi_i(p_j) + \frac{db_{i,k}}{dt}\psi_i(p_j) = f_k\left(p_j, t, \hat{u}_1(p_j), \ldots, \hat{u}_M(p_j), \ldots, (\hat{u}_1)_{xx}(p_j), \ldots, (\hat{u}_M)_{xx}(p_j)\right)$$

for $k = 1, \ldots, M$ and $j = 1, \ldots, 2(N-1)$.

This is a system of $2M(N-1)$ ordinary differential equations in $2M\,N$ unknown coefficient functions, $a_{i,k}$ and $b_{i,k}$. This system can be written in the matrix−vector form as $A\,dc/dt = F(t, y)$ with $c(t_0) = c_0$ where c is a vector of coefficients of length $2M\,N$ and c_0 holds the initial values of the coefficients. The last $2M$ equations are obtained by differentiating the boundary conditions

$$\alpha_k \frac{da_k}{dt} + \beta_k \frac{db_k}{dt} = \frac{d\gamma_k}{dt}$$

for $k = 1, \ldots, M$.

The initial conditions $u_k(x, t_0)$ must satisfy the boundary conditions. Also, the $\gamma_k(t)$ must be continuous and have a smooth derivative, or the boundary conditions will not be properly imposed for $t > t_0$.

If $\alpha_k = \beta_k = 0$, it is assumed that no boundary condition is desired for the k-th unknown at the left endpoint. A similar comment holds for the right endpoint. Thus, collocation is done at the endpoint. This is generally a useful feature for systems of first-order partial differential equations.

If the number of partial differential equations is $M = 1$ and the number of breakpoints is $N = 4$, then

$$A = \begin{bmatrix} \alpha_1 & \beta_1 & & & & & & \\ \phi_1(p_1) & \psi_1(p_1) & \phi_2(p_1) & \psi_2(p_1) & & & & \\ \phi_1(p_2) & \psi_1(p_2) & \phi_2(p_2) & \psi_2(p_2) & & & & \\ & & \phi_3(p_3) & \psi_3(p_3) & \phi_4(p_3) & \psi_4(p_3) & & \\ & & \phi_3(p_4) & \psi_3(p_4) & \phi_4(p_4) & \psi_4(p_4) & & \\ & & & & \phi_5(p_5) & \psi_5(p_5) & \phi_6(p_5) & \psi_6(p_5) \\ & & & & \phi_5(p_6) & \psi_5(p_6) & \phi_6(p_6) & \psi_6(p_6) \\ & & & & & & \alpha_4 & \beta_4 \end{bmatrix}$$

The vector c is

$$c = [a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_3]^T$$

and the right-side F is

$$F = \left[\gamma'(x_1),\ f(p_1),\ f(p_2),\ f(p_3),\ f(p_4),\ f(p_5),\ f(p_6),\ \gamma'(x_4)\right]^T$$

If $M > 1$, then each entry in the above matrix is replaced by an $M \times M$ diagonal matrix. The element α_1 is replaced by $\text{diag}(\alpha_{1,1}, \ldots, \alpha_{1,M})$. The elements α_N, β_1 and β_N are handled in the same manner. The $\phi_i(p_j)$ and $\psi_i(p_j)$ elements are replaced by $\phi_i(p_j)I_M$ and $\psi_i(p_j)I_M$ where I_M is the identity matrix of order M. See Madsen and Sincovec (1979) for further details about discretization errors and Jacobian matrix structure.

The input/output array `Y` contains the values of the $a_{k,i}$. The initial values of the $b_{k,i}$ are obtained by using the IMSL cubic spline routine `imsl_f_cub_spline_interp_e_cnd` (Chapter 3, "Interpolation and Approximation") to construct functions

$$\hat{u}_k(x, t_0)$$

such that

$$\hat{u}_k(x_i, t_0) = a_{ki}$$

The IMSL routine `imsl_f_cub_spline_value`, Chapter 3, "Interpolation and Approximation" is used to approximate the values

$$\frac{d\hat{u}_k}{dx}(x_i, t_0) \equiv b_{k,i}$$

There is an optional use of `imsl_f_pde_method_of_lines` that allows the user to provide the initial values of $b_{k,i}$.

The order of matrix A is $2MN$ and its maximum bandwidth is $6M - 1$. The band structure of the Jacobian of F with respect to c is the same as the band structure of A. This system is solved using a modified version of `imsl_f_ode_adams_gear`. Some of the linear solvers were removed. Numerical Jacobians are used exclusively. The algorithm is unchanged. Gear's BDF method is used as the default because the system is typically stiff.

Four examples of PDEs are now presented that illustrate how users can interface their problems with IMSL PDE solving software. The examples are small and not indicative of the complexities that most practitioners will face in their applications. A set of seven sample application problems, some of them with more than one equation, is given in Sincovec and Madsen (1975). Two further examples are given in Madsen and Sincovec (1979).

Examples

Example 1

The normalized linear diffusion PDE, $u_t = u_{xx}$, $0 \le x \le 1$, $t > t_0$, is solved. The initial values are $t_0 = 0$, $u(x, t_0) = u_0 = 1$. There is a "zero-flux" boundary condition at $x = 1$, namely $u_x(1, t) = 0$, $(t > t_0)$. The boundary value of $u(0, t)$ is abruptly changed from u_0 to the value $u_1 = 0.1$. This transition is completed by $t = t_\delta = 0.09$.

Due to restrictions in the type of boundary conditions successfully processed by `imsl_f_pde_method_of_lines`, it is necessary to provide the derivative boundary value function γ' at $x = 0$ and at $x = 1$. The function γ at $x = 0$ makes a smooth transition from the value u_0 at $t = t_0$ to the value u_1 at $t = t_\delta$. The transition phase for γ' is computed by evaluating a cubic interpolating polynomial. For this purpose, the function subprogram `imsl_f_cub_spline_value`, Chapter 3, "Interpolation and Approximation" is used. The interpolation is performed as a first step in the user-supplied routine `fcn_bc`. The function and derivative values $\gamma(t_0) = u_0$, $\gamma'(t_0) = 0$, $\gamma(t_\delta) = u_1$, and $\gamma'(t_\delta) = 0$, are used as input to routine `imsl_f_cub_spline_interp_e_cnd`, to obtain the coefficients evaluated by `imsl_f_cub_spline_value`. Notice that $\gamma'(t) = 0$, $t > t_\delta$. The evaluation routine `imsl_f_cub_spline_value` will not yield this value so logic in the routine `fcn_bc` assigns $\gamma'(t) = 0$, $t > t_\delta$.

Link to example source (pde_method_of_lines_ex1.c)

```
#include <imsl.h>
#include <math.h>

int main() {
    void fcnut(int, float, float, float *, float *, float *,
        float *);
    void fcnbc(int, float, float, float *, float *,
        float *);
    int npdes=1, nx=8, i, j=1, nstep=10;
    float t=0.0, tend, xbreak[8], y[8];
    char title[50];
    void *state;

    /* Set breakpoints and initial conditions */
    for (i = 0; i < nx; i++) {
```

```
        xbreak[i] = (float) i / (float) (nx - 1);
        y[i] = 1.0;
    }

    /* Initialize the solver */
    imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
        IMSL_TOL, sqrt(imsl_f_machine(4))*1000.0,
        IMSL_HINIT, (0.1*sqrt(imsl_f_machine(4))*1000.0),
        0);

    while (j <= nstep) {
        tend = (float) j++ / (float) nstep;
        tend *= tend;

        /* Solve the problem */
        imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
            state, fcnut, fcnbc);

        /* Print results at current t=tend */
        sprintf(title, "solution at t = %4.2f\0", t);
        imsl_f_write_matrix(title, npdes, nx, y, 0);
    }
}


void fcnut(int npdes, float x, float t, float *u, float *ux,
    float *uxx, float *ut) {
    /* Define the PDE */
    *ut = *uxx;
}


void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
    float *gamp) {
    static int ndata, first = 1;
    static float delta=0.09, u0=1.0, u1 = 0.1;
    static float dfdata[2], xdata[2], fdata[2];
    static Imsl_f_ppoly *ppoly;

    /* Compute interpolant first time only */
    if (first) {
        first = 0;
        ndata = 2;
        xdata[0] = 0.0;
        xdata[1] = delta;
        fdata[0] = u0;
        fdata[1] = u1;
        dfdata[0] = dfdata[1] = 0.0;
        ppoly = imsl_f_cub_spline_interp_e_cnd(ndata, xdata, fdata,
            IMSL_LEFT, 1, dfdata[0],
            IMSL_RIGHT, 1, dfdata[1],
            0);
    }

    /* Define boundary conditions */
    if (x == 0.0) {
        /* These are for x = 0 */
```

```
        *alpha = 1.0;
        *beta = 0.0;
        *gamp = 0.0;

        /* If in the boundary layer, compute
        nonzero gamma prime */
        if (t <= delta)
            *gamp = imsl_f_cub_spline_value(t, ppoly, IMSL_DERIV, 1, 0);
    } else {
        /* These are for x = 1 */
        *alpha = 0.0;
        *beta = 1.0;
        *gamp = 0.0;
    }
}
```

Output

```
                        solution at t = 0.01
      1           2           3           4           5           6
  0.959       0.993       0.999       1.000       1.000       1.000

      7           8
  1.000       1.000

                        solution at t = 0.04
      1           2           3           4           5           6
 0.6347      0.8485      0.9371      0.9730      0.9877      0.9940

      7           8
 0.9966      0.9974

                        solution at t = 0.09
      1           2           3           4           5           6
 0.3307      0.5965      0.7662      0.8676      0.9255      0.9569

      7           8
 0.9722      0.9768

                        solution at t = 0.16
      1           2           3           4           5           6
 0.3307      0.4933      0.6357      0.7477      0.8282      0.8809

      7           8
 0.9102      0.9196

                        solution at t = 0.25
      1           2           3           4           5           6
 0.3307      0.4513      0.5626      0.6576      0.7324      0.7856

      7           8
 0.8171      0.8276

                        solution at t = 0.36
      1           2           3           4           5           6
 0.3307      0.4216      0.5070      0.5824      0.6440      0.6894
```

```
     7           8
0.7171      0.7265

                    solution at t = 0.49
     1           2           3           4           5           6
0.3307      0.4020      0.4694      0.5295      0.5792      0.6162

     7           8
0.6391      0.6468

                    solution at t = 0.64
     1           2           3           4           5           6
0.3307      0.3896      0.4454      0.4951      0.5364      0.5671

     7           8
0.5861      0.5925

                    solution at t = 0.81
     1           2           3           4           5           6
0.3307      0.3756      0.4182      0.4563      0.4878      0.5115

     7           8
0.5261      0.5310

                    solution at t = 1.00
     1           2           3           4           5           6
0.3307      0.3620      0.3918      0.4184      0.4406      0.4573

     7           8
0.4677      0.4712
```

Example 2

Here, Problem C is solved from Sincovec and Madsen (1975). The equation is of diffusion-convection type with discontinuous coefficients. This problem illustrates a simple method for programming the evaluation routine for the derivative, u_t. Note that the weak discontinuities at $x = 0.5$ are not evaluated in the expression for u_t. The problem is defined as

$$u_t = \partial u / \partial t = \partial / \partial x\left(D(x)\partial u / \partial x\right) - v(x)\partial u / \partial x$$

$$x \in [0, 1], t > 0$$

$$D(x) = \begin{cases} 5 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$v(x) = \begin{cases} 1000.0 & \text{if } 0 \le x < 0.5 \\ 1 & \text{if } 0.5 < x \le 1.0 \end{cases}$$

$$u(x, 0) = \begin{cases} 1 \text{ if } x = 0 \\ 0 \text{ if } x > 0 \end{cases}$$

$$u(0, t) = 1, \quad u(1, t) = 0$$

Link to example source (pde_1d_mg_ex2.c)

```
#include <imsl.h>
#include <math.h>

int main()
{
        void            fcnut(int, float, float, float *, float *, float *,
                              float *);
        void            fcnbc(int, float, float, float *, float *,
                              float *);
        int             npdes = 1;
        int             nx = 100;
        int             i;
        int             j = 1;
        int             nstep = 10;
        float           t = 0.0;
        float           tend;
        float           xbreak[100];
        float           y[100];
        float           tol, hinit;
        char            title[50];
        void           *state;

                        /* Set breakpoints and initial conditions */

        for (i = 0; i < nx; i++) {
                xbreak[i] = (float) i / (float) (nx - 1);
                y[i] = 0.0;
        }
        y[0] = 1.0;

                        /* Initialize the solver */

        tol = sqrt(imsl_f_machine(4));
        hinit = 0.01*tol;
        imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
                                       IMSL_TOL, tol,
                                       IMSL_HINIT, hinit,
                                       0);

        while (j <= nstep) {
                tend = (float) j++ / (float) nstep;

                        /* Solve the problem */

                imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
                                           state, fcnut, fcnbc);
        }
                        /* Print results at t=tend */

                sprintf(title, "solution at t = %4.2f\0", t);
                imsl_f_write_matrix(title, npdes, nx, y, 0);
}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
```

```
        float *ut)
{
                        /* Define the PDE */

        float v;
        float d;


        if (x <= 0.5) {
                d = 5.0;
                v = 1000.0;
        }
        else
                d = v = 1.0;

        ut[0] = d*uxx[0] - v*ux[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
        float *gamp)
{
        *alpha = 1.0;
        *beta = 0.0;
        *gamp = 0.0;
}
```

Output

```
                    solution at t = 1.00
     1           2           3           4           5           6
 1.000       1.000       1.000       1.000       1.000       1.000

     7           8           9          10          11          12
 1.000       1.000       1.000       1.000       1.000       1.000

    13          14          15          16          17          18
 1.000       1.000       1.000       1.000       1.000       1.000

    19          20          21          22          23          24
 1.000       1.000       1.000       1.000       1.000       1.000

    25          26          27          28          29          30
 1.000       1.000       1.000       1.000       1.000       1.000

    31          32          33          34          35          36
 1.000       1.000       1.000       1.000       1.000       1.000

    37          38          39          40          41          42
 1.000       1.000       1.000       1.000       1.000       1.000

    43          44          45          46          47          48
 1.000       1.000       1.000       1.000       1.000       1.000

    49          50          51          52          53          54
 1.000       0.997       0.984       0.969       0.953       0.937
```

```
    55          56          57          58          59          60
 0.921       0.905       0.888       0.872       0.855       0.838

    61          62          63          64          65          66
 0.821       0.804       0.786       0.769       0.751       0.733

    67          68          69          70          71          72
 0.715       0.696       0.678       0.659       0.640       0.621

    73          74          75          76          77          78
 0.602       0.582       0.563       0.543       0.523       0.502

    79          80          81          82          83          84
 0.482       0.461       0.440       0.419       0.398       0.376

    85          86          87          88          89          90
 0.354       0.332       0.310       0.288       0.265       0.242

    91          92          93          94          95          96
 0.219       0.196       0.172       0.148       0.124       0.100

    97          98          99         100
 0.075       0.050       0.025       0.000
```

Example 3

In this example, using `imsl_f_pde_method_of_lines`, the linear normalized diffusion PDE $u_t = u_{xx}$ is solved but with an optional use that provides values of the derivatives, u_x, of the initial data. Due to errors in the numerical derivatives computed by spline interpolation, more precise derivative values are required when the initial data is $u(x, 0) = 1 + \cos[(2n - 1)\pi x]$, $n > 1$. The boundary conditions are "zero flux" conditions $u_x(0, t) = u_x(1, t) = 0$ for $t > 0$. Note that the initial data is compatible with these end conditions since the derivative function

$$u_x(x,0) = \frac{du(x,0)}{dx} = -(2n-1)\pi \sin\left[(2n-1)\pi x\right]$$

vanishes at $x = 0$ and $x = 1$.

This optional usage signals that the derivative of the initial data is passed by the user. The values $u(x, tend)$ and $u_x(x, tend)$ are output at the breakpoints with the optional usage.

Link to example source (pde_method_of_lines_ex3.c)

```
#include <imsl.h>
#include <math.h>

int main()
{
        void            fcnut(int, float, float, float *, float *, float *,
                              float *);
        void            fcnbc(int, float, float, float *, float *, float *);
        int             npdes = 1;
        int             nx = 10;
        int             i;
```

```
        int             j = 1;
        int             nstep = 10;
        float           t = 0.0;
        float           tend = 0.0;
        float           xbreak[10];
        float           y[10], deriv[10];
        float           tol, hinit;
        float           pi, arg;
        char            title1[50];
        char            title2[50];
        void           *state;

        pi = imsl_d_constant("pi", 0);
        arg = 9.0 * pi;

                        /* Set breakpoints and initial conditions */

        for (i = 0; i < nx; i++) {
                xbreak[i] = (float) i / (float) (nx - 1);
                y[i] = 1.0 + cos(arg * xbreak[i]);
                deriv[i] = -arg * sin(arg * xbreak[i]);
        }

                        /* Initialize the solver */

        tol = sqrt(imsl_f_machine(4));
        imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
                                       IMSL_TOL, tol,
                                       IMSL_INITIAL_VALUE_DERIVATIVE,
                                       deriv,
                                       0);

        while (j <= nstep) {
                j++;
                tend += 0.001;

                        /* Solve the problem */

                imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
                                           state, fcnut, fcnbc);

                        /* Print results at at every other t=tend */

                if (j % 2) {
                        sprintf(title1, "\nsolution at t = %5.3f\0", t);
                        sprintf(title2, "\nderivative at t = %5.3f\0", t);
                        imsl_f_write_matrix(title1, npdes, nx, y, 0);
                        imsl_f_write_matrix(title2, npdes, nx, deriv, 0);
                }
        }

}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
      float *ut)
{
                        /* Define the PDE */
```

```
        ut[0] = uxx[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
      float *gamp)
{
                        /* Define the boundary conditions */

        alpha[0] = 0.0;
        beta[0] = 1.0;
        gamp[0] = 0.0;
}
```

Output

```
                      solution at t = 0.002
        1           2           3           4           5           6
    1.233       0.767       1.233       0.767       1.233       0.767

        7           8           9          10
    1.233       0.767       1.233       0.767


                     derivative at t = 0.002
        1           2           3           4           5           6
 0.000e+00  -5.172e-07   1.911e-06   1.818e-06  -5.230e-07   2.408e-06

        7           8           9          10
-2.517e-06   3.194e-06  -3.608e-06   2.023e-06


                      solution at t = 0.004
        1           2           3           4           5           6
    1.053       0.947       1.053       0.947       1.053       0.947

        7           8           9          10
    1.053       0.947       1.053       0.947


                     derivative at t = 0.004
        1           2           3           4           5           6
 0.000e+00  -1.332e-06  -9.059e-06  -4.401e-06   5.006e-06  -2.134e-06

        7           8           9          10
-1.733e-06   4.625e-06   6.741e-07   2.023e-06



                      solution at t = 0.006
        1           2           3           4           5           6
    1.012       0.988       1.012       0.988       1.012       0.988

        7           8           9          10
    1.012       0.988       1.012       0.988
```

```
                    derivative at t = 0.006
        1           2           3           4           5           6
0.000e+00  -1.408e-06  -1.018e-06  -6.572e-07  -8.213e-07  -1.151e-06

        7           8           9          10
1.051e-06   1.257e-06  -2.920e-07   2.023e-06


                     solution at t = 0.008
        1           2           3           4           5           6
    1.003       0.997       1.003       0.997       1.003       0.997

        7           8           9          10
    1.003       0.997       1.003       0.997


                    derivative at t = 0.008
        1           2           3           4           5           6
0.000e+00  -1.028e-06   4.270e-06   3.114e-06  -3.085e-06  -1.492e-06

        7           8           9          10
2.126e-06  -1.280e-06  -1.541e-06   2.023e-06


                     solution at t = 0.010
        1           2           3           4           5           6
    1.001       0.999       1.001       0.999       1.001       0.999

        7           8           9          10
    1.001       0.999       1.001       0.999


                    derivative at t = 0.010
        1           2           3           4           5           6
0.000e+00  -7.596e-07   2.819e-07   1.547e-07  -1.469e-06  -9.516e-07

        7           8           9          10
2.889e-07   8.956e-08   5.992e-07   2.023e-06
```

Example 4

In this example, consider the linear normalized hyperbolic PDE, $u_{tt} = u_{xx}$, the "vibrating string" equation. This naturally leads to a system of first order PDEs. Define a new dependent variable $u_t = v$. Then, $v_t = u_{xx}$ is the second equation in the system. Take as initial data $u(x, 0) = \sin(\pi x)$ and $u_t(x, 0) = v(x, 0) = 0$. The ends of the string are fixed so $u(0, t) = u(1, t) = v(0, t) = v(1, t) = 0$. The exact solution to this problem is $u(x, t) = \sin(\pi x) \cos(\pi t)$. Residuals are computed at the output values of t for $0 < t \leq 2$. Output is obtained at 200 steps in increments of 0.01.

Even though the sample code `imsl_f_pde_method_of_lines` gives satisfactory results for this PDE, users should be aware that for *nonlinear problems*, "shocks" can develop in the solution. The appearance of shocks may cause the code to fail in unpredictable ways. See Courant and Hilbert (1962), pp 488-490, for an introductory discussion of shocks in hyperbolic systems.

Link to example source (pde_method_of_lines_ex4.c)

```
#include <imsl.h>
#include <math.h>

int main()
{
      void            fcnut(int, float, float, float *, float *, float *,
                            float *);
      void            fcnbc(int, float, float, float *, float *, float *);
      int             npdes = 2;
      int             nx = 10;
      int             i;
      int             j = 1;
      int             nstep = 200;
      float           t = 0.0;
      float           tend = 0.0;
      float           xbreak[20];
      float           y[20], deriv[20];
      float           tol, hinit;
      float           pi;
      float           error[10], erru;
      void           *state;

      pi = imsl_d_constant("pi", 0);

                      /* Set breakpoints and initial conditions */

      for (i = 0; i < nx; i++) {
              xbreak[i] = (float) i / (float) (nx - 1);
              y[i] = sin(pi * xbreak[i]);
              y[nx + i] = 0.0;
              deriv[i] = pi * cos(pi * xbreak[i]);
              deriv[nx + i] = 0.0;
      }

                      /* Initialize the solver */

      tol = sqrt(imsl_f_machine(4));
      imsl_f_pde_method_of_lines_mgr(IMSL_PDE_INITIALIZE, &state,
                                     IMSL_TOL, tol,
                                     IMSL_INITIAL_VALUE_DERIVATIVE,
                                     deriv,
                                     0);

      while (j <= nstep) {
              j++;
              tend += 0.01;
                      /* Solve the problem */

              imsl_f_pde_method_of_lines(npdes, &t, tend, nx, xbreak, y,
                                         state, fcnut, fcnbc);


                      /* Look at output at steps of 0.01
                         and compute errors */
```

```
            for (i = 0; i < nx; i++) {
                    error[i] = y[i] - sin(pi * xbreak[i]) *
                                      cos(pi *tend);
                    erru = imsl_f_max(erru, fabs(error[i]));
                  }
          }
        printf("Maximum error in u(x,t) = %e\n", erru);

}

void fcnut(int npdes, float x, float t, float *u, float *ux, float *uxx,
           float *ut)
{
                      /* Define the PDE */

        ut[0] = u[1];
        ut[1] = uxx[0];
}

void fcnbc(int npdes, float x, float t, float *alpha, float *beta,
      float *gamp)
{
                     /* Define the boundary conditions */

        alpha[0] = 1.0;
        beta[0] = 0.0;
        gamp[0] = 0.0;
        alpha[1] = 1.0;
        beta[1] = 0.0;
        gamp[1] = 0.0;
}
```

Output

```
Maximum error in u(x,t) = 6.228203e-04
```

feynman_kac

This function solves the generalized Feynman-Kac PDE on a rectangular grid using a finite element Galerkin method. Initial and boundary conditions are satisfied. The solution is represented by a series of C^2 Hermite quintic splines.

Synopsis

#include `<imsl.h>`

void `imsl_f_feynman_kac` (*int* `nxgrid`, *int* `ntgrid`, *int* `nlbcd`, *int* `nrbcd`, *float* `xgrid[]`, *float* `tgrid[]`, *void* `fcn_fkcfiv()`, *void* `fcn_fkbcp()`, *float* `y[]`, *float* `y_prime[]`,..., 0)

The type *double* function is `imsl_d_feynman_kac`.

Required Arguments

int `nxgrid` (Input)
Number of grid lines in the *x*-direction. `nxgrid` must be at least 2.

int `ntgrid` (Input)
Number of time points where an approximate solution is returned. The value `ntgrid` must be at least 1.

int `nlbcd` (Input)
Number of left boundary conditions. It is required that $1 \leq$ `nlbcd` ≤ 3.

int `nrbcd` (Input)
Number of right boundary conditions. It is required that $1 \leq$ `nrbcd` ≤ 3.

float `xgrid[]` (Input)
Array of length `nxgrid` containing the breakpoints for the Hermite quintic splines used in the *x* discretization. The points in `xgrid` must be strictly increasing. The values `xgrid[0]` and `xgrid[nxgrid-1]` are the endpoints of the interval.

float `tgrid[]` (Input)
Array of length `ntgrid` containing the set of time points (in time-remaining units) where an approximate solution is returned. The points in `tgrid` must be positive and strictly increasing.

void `fcn_fkcfiv` (*float* `x`, *float* `t`, *int* `*iflag`, *float* `*value`)
User-supplied function to compute the values of the coefficients $\sigma, \sigma', \mu, \kappa$ for the Feynman-Kac PDE and the initial data function $p(x)$, $x_{min} £ x £ x_{max}$.

float `x` (Input)
Space variable.

float `t` (Input)
Time variable.

int `*iflag` (Input/Output)
On entry, `iflag` indicates which coefficient or data function is to be computed. The following table shows which value has to be returned by `fcn_fkcfiv` for all possible values of `iflag`:

iflag	Computed coefficien

iflag	Computed coefficien
-1	$\sigma' = \dfrac{\partial \sigma(x,t)}{\partial x}$
0	$p(x)$
1	σ
2	μ
3	κ

For non-zero input values of `iflag`, note when a coefficient does not depend on *t*. This is done by setting `iflag` = 0 after the coefficient is defined. If there is time dependence, the value of `iflag` should not be changed. This action will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each *t* value.

float `*value` (Output)
Value of the coefficient or initial data function. Which value is computed depends on the input value for `iflag`, see description of `iflag`.

void `fcn_fkbcp` (*int* `nbc`, *float* `t`, *int* `*iflag`, *float* `values[]`)
User-supplied function to define boundary values that the solution of the differential equation must satisfy. There are `nlbcd` conditions specified at the left end, $x_{\min}$, and `nrbcd` conditions at the right end, $x_{\max}$. The boundary conditions are

$$a(x,t)\,f + b(x,t)\,f_x + c(x,t)\,f_{xx} = d(x,t), \quad x = x_{\min} \text{ or } x = x_{\max}.$$

int `nbc` (Input)
Number of boundary conditions. At x_{min}, `nbc=nlbcd`, at x_{max}, `nbc` = `nrbcd`.

float `t` (Input)
Time point of the boundary conditions.

int `*iflag` (Input/Output)
On entry, `iflag` indicates whether the coefficients for the left or right boundary conditions are to be computed:

iflag	Computed boundary conditions
1	Left end, x = x_{min}
0	Right end, x = x_{max}

If there is no time dependence for one of the boundaries then set `iflag` = 0 after the array `values` is defined for either end point. This flag can avoid unneeded continued computation of the finite element matrices.

float `values[]` (Output)

Array of length `4*max(nlbcd,nrbcd)` containing the values of the boundary condition coefficients in its first `4*nbc` locations. The coefficients for x_{min} are stored row-wise according to the following scheme:

$$\begin{pmatrix} a_1(x_{\min},t), b_1(x_{\min},t), c_1(x_{\min},t), d_1(x_{\min},t) \\ \vdots \\ a_{nlbcd}(x_{\min},t), b_{nlbcd}(x_{\min},t), c_{nlbcd}(x_{\min},t), d_{nlbcd}(x_{\min},t) \end{pmatrix}.$$

The coefficients for x_{max} are stored similarly.

float `y[]` (Output)

An array of size (`ntgrid`+1) by (`3*nxgrid`) containing the coefficients of the Hermite representation of the approximate solution for the Feynman-Kac PDE at time points 0, `tgrid[0]`, …, `tgrid[ntgrid-1]`. The approximate solution is given by

$$f(x,t) = \sum_{j=0}^{3*\texttt{nxgrid}-1} y_{ij}\,\beta_j(x) \quad \text{for } t = \texttt{tgrid}[i-1],\ i = 1,\ldots,\texttt{ntgrid}.$$

The representation for the initial data at $t = 0$ is

$$p(x) = \sum_{j=0}^{3*\texttt{nxgrid}-1} y_{0j}\,\beta_j(x).$$

The (`ntgrid`+1) by (`3*nxgrid`) matrix

$$(y_{ij})_{i=0,\ldots,\texttt{ntgrid}}^{j=0,\ldots,3*\texttt{nxgrid}-1}$$

is stored row-wise in array `y`.

After the integration, use row `i` of array `y` as input argument `coef` in function `imsl_f_feynman_kac_evaluate` to obtain an array of values for $f(x, t)$ or its partials f_x, f_{xx}, f_{xxx} at time point `t=0`,`tgrid[i-1]`, `i=1`,…,`ntgrid`.

The expressions for the basis functions $\beta_j(x)$ are represented piece-wise and can be found in Hanson, R. (2008) "Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements".

float `y_prime[]` (Output)
An array of size (`ntgrid` + 1) by (3`*nxgrid`) containing the first derivatives of `y` at time points 0, `tgrid[0]`,…,`tgrid[ntgrid - 1]`, i.e.

$$f_t\left(x,\bar{t}\right) = \sum_{j=0}^{3*\texttt{nxgrid-1}} y'_{ij}\beta_j\left(x\right) \quad \text{for} \quad \bar{t} = \texttt{tgrid}[i-1],\ i = 1,\ldots,\texttt{ntgrid},$$

and

$$f_t\left(x,\bar{t}\right) = \sum_{j=0}^{3*\texttt{nxgrid}-1} y'_{0j}\beta_j\left(x\right) \quad \text{for } \bar{t} = 0.$$

The (`ntgrid` + 1) by (3`*nxgrid`) matrix

$$(y'_{ij})_{i=0,\ldots,\texttt{ntgrid}}^{j=0,\ldots,3*\texttt{nxgrid}-1}$$

is stored row-wise in array `y_prime`.

After the integration, use row `i` of array `y_prime` as input argument `coef` in function `imsl_f_feynman_kac_evaluate` to obtain an array of values for the partials $f_t, f_{tx}, f_{txx}, f_{txxx}$ at time point `t=tgrid[i-1]`, `i=1`,…,`ntgrid`, and row `0` for the partials at `t=0`.

Synopsis with Optional Arguments

```
#include <imsl.h>
```

void `imsl_f_feynman_kac` (*int* `nxgrid`, *int* `ntgrid`, *int* `nlbcd`, *int* `nrbcd`,
float `xgrid[]`, *float* `tgrid[]`, *void* `fcn_fkcfiv()`,
void `fcn_fkbcp()`,*float* `y[]`, *float* `y_prime[]`,
`IMSL_FCN_FKCFIV_W_DATA`, *void* `fcn_fkcfiv()`, *void* `*data`,
`IMSL_FCN_FKBCP_W_DATA`, *void* `fcn_fkbcp()`, *void* `*data`,

IMSL_FCN_INIT, *void* fcn_fkinit(),
IMSL_FCN_INIT_W_DATA, *void* fcn_fkinit(), *void* *data,
IMSL_FCN_FORCE, *void* fcn_fkforce(),
IMSL_FCN_FORCE_W_DATA, *void* fcn_fkforce(), *void* *data,
IMSL_ATOL_RTOL_SCALARS, *float* atol, *float* rtol,
IMSL_ATOL_RTOL_ARRAYS, *float* atol[], *float* rtol[]
IMSL_NDEGREE, *int* ndeg,
IMSL_TDEPEND, *int* tdepend[],
IMSL_MAX_STEP, *float* max_stepsize,
IMSL_INITIAL_STEPSIZE, *float* init_stepsize,
IMSL_MAX_NUMBER_STEPS, *int* max_steps,
IMSL_STEP_CONTROL, *int* step_control,
IMSL_MAX_BDF_ORDER, *int* max_bdf_order,
IMSL_T_BARRIER, *float* t_barrier,
IMSL_ISTATE, *int* istate[],
IMSL_EVALS, *int* nval[],
0)

Optional Arguments

IMSL_FCN_FKCFIV_W_DATA,
void fcn_fkcfiv(*float* x, *float* t, *int* *iflag, *float* *value, *void* *data),
void *data (Input)
User-supplied function to compute the values of the coefficients $\sigma, \sigma', \mu, \kappa$ for the Feynman-Kac PDE and the initial data function $p(x)$, $x_{\min} \le x \le x_{\max}$. This function also accepts a pointer to the object data supplied by the user.

IMSL_FCN_FKBCP_W_DATA,
void fcn_fkbcp(*int* nbc, *float* t, *int* *iflag, *float* values[], *void* *data) , *void* *data (Input)
User-supplied function to define boundary values that the solution of the differential equation must satisfy. This function also accepts a pointer to data supplied by the user.

IMSL_FCN_INIT, *void* fcn_fkinit(*int* nxgrid, *int* ntgrid, *float* xgrid[], *float* tgrid[], *float* time, *float* yprime[], *float* y[], *float* atol[], *float* rtol[]) (Input)
User-supplied function for adjustment of initial data or as an opportunity for output during the integration steps. The solution values of the model parameters are presented in the arrays y and yprime at the integration points time = 0, tgrid[j], j=0,…,ntgrid-1. At the initial point, integral least-squares estimates are made for representing the initial data $p(x)$. If this is not satisfactory, fcn_fkinit can change the contents of y[] to match data for any reason.

int nxgrid (Input)
Number of grid lines in the *x*-direction.

int `ntgrid` (Input)
Number of time points where an approximate solution is returned.

float `xgrid[]` (Input)
Vector of length `nxgrid` containing the breakpoints for the Hermite quintic splines used in the *x* discretization.

float `tgrid[]` (Input)
Vector of length `ntgrid` containing the set of time points (in time-remaining units) where an approximate solution is returned.

float `time` (Input)
Time variable.

float `yprime[]` (Input)
Vector of length `3*nxgrid` containing the derivative of the coefficients of the Hermite quintic spline at time point `time`.

float `y[]` (Input/Output)
Vector of length `3*nxgrid` containing the coefficients of the Hermite quintic spline at time point `time`.

float `atol[]` (Input/Output)
Array of length `3*nxgrid` containing absolute error tolerances.

float `rtol[]` (Input/Output)
Array of length `3*nxgrid` containing relative error tolerances.

`IMSL_FCN_INIT_W_DATA`, *void* fcn_fkinit(*int* `nxgrid`, *int* `ntgrid`, *float* `xgrid[]`, *float* `tgrid[]`, *float* `time`, *float* `yprime[]`, *float* `y[]`, *float* `atol[]`, *float* `rtol[]`, *void* `*data`), *void* `*data` (Input)
User-supplied function for adjustment of initial data or as an opportunity for output during the integration steps which also accepts a pointer to data supplied by the user. For an explanation of the other arguments of function `fcn_fkinit`, see optional argument `IMSL_FCN_INIT`.

`IMSL_FCN_FORCE`, *void* `fcn_fkforce`(*int* `interval`, *int* `ndeg`, *int* `nxgrid`, *float* `y[]`, *float* `time`, *float* `width`, *float* `xlocal[]`, *float* `qw[]`, *float* `u[]`, *float* `phi[]`, *float* `dphi[]`) (Input)
Function `fcn_fkforce` is used in case there is a non-zero term $\phi(f, x, t)$ in the Feynman-Kac differential equation. If function `fcn_fkforce` is not used, it is assumed that $\phi(f, x, t)$ is identically zero.

int `interval` (Input)
Index indicating the bounds `xgrid[interval-1]` and `xgrid[interval]` of the integration interval, $1 \leq$ `interval` $\leq$ `nxgrid`-1.

int `ndeg` (Input)
The degree used for the Gauss-Legendre formulas.

int `nxgrid` (Input)
Number of grid lines in the x-direction.

float `y[]` (Input)
Vector of length `3*nxgrid` containing the coefficients of the Hermite quintic spline representing the solution of the Feynman-Kac equation at time point `time`.

float `time` (Input)
Time variable.

float `width` (Input)
The interval width, `width` = `xgrid[interval] - xgrid[interval-1]`.

float `xlocal[]` (Input)
Array of length `ndeg` containing the Gauss-Legendre points translated and normalized to the interval `[xgrid[interval-1], xgrid[interval]]`.

float `qw[]` (Input)
Vector of length `ndeg` containing the Gauss-Legendre weights.

float `u[]` (Input)
Array of dimension 12 by `ndeg` containing the basis function values that define $\tilde{\beta}(x)$ at the Gauss-Legendre points `xlocal`. Setting $u_{k,i} :=$ `u[i+k*ndeg]` and $x_i :=$ `xlocal[i]`, $\tilde{\beta}(x_i)$ is defined as

$$\tilde{\beta}(x_i) := (\beta_{3*(\texttt{interval-1})}(x_i), \ldots, \beta_{3*\texttt{interval}+2}(x_i))^T = (u_{0,i}, u_{1,i}, u_{2,i}, u_{6,i}, u_{7,i}, u_{8,i})^T.$$

float `phi[]` (Output)
Vector of length 6 containing Gauss-Legendre approximations for the local contributions

$$\varphi_t := \int_{\texttt{xgrid[interval-1]}}^{\texttt{xgrid[interval]}} \phi(f,x,t)\tilde{\beta}(x)dx,$$

where $t =$ time and

$$\tilde{\beta}(x) := (\beta_{3*(\texttt{interval-1})}(x),\ldots,\beta_{3*\texttt{interval}+2}(x))^T.$$

Vector phi contains elements

$$\texttt{phi[i]} = \textit{width} * \sum_{j=0}^{n\deg GL-1} \texttt{qw[j]}\ \tilde{\beta}_i(x_j)\ \phi(f,\texttt{xlocal[j]},\text{time}),$$

for i=0,…,5.

float dphi[] (Output)
Array of dimension 6 by 6 containing a Gauss-Legendre approximation for the Jacobian of the local contributions φ_t at $t =$ time,

$$\frac{\partial \varphi_t}{\partial y} = \int_{\texttt{xgrid[interval-1]}}^{\texttt{xgrid[interval]}} \frac{\partial \phi(f,x,t)}{\partial f}\tilde{\beta}(x)\tilde{\beta}^T(x)dx.$$

The approximation to this symmetric matrix is stored row-wise, i.e.

$$\texttt{dphi[j+i*6]} = \textit{width}\ * \sum_{k=0}^{n\deg GL-1} \texttt{qw[k]}\ \tilde{\beta}_i(x_k)\ \tilde{\beta}_j(x_k)\ \left.\frac{\partial \phi}{\partial f}\right|_{x=xlocal[k],t=time}$$

for i, j = 0, …, 5.

IMSL_FCN_FORCE_W_DATA,
void fcn_fkforce(*int* interval, *int* ndeg, *int* nxgrid, *float* y[], *float* time, *float* width, *floa* xlocal[], *float* qw[], *float* u[], *float* phi[], *float* dphi[], *void* *data), *void* *data (Input)
Function fcn_fkforce is used in case there is a non-zero term $\phi(f,x,t)$ in the Feynman-Kac differential equation. This function also accepts a data pointer to data supplied by the user. For an explanation of the other arguments of function fcn_fkforce, see optional argument IMSL_FCN_FORCE.

IMSL_ATOL_RTOL_SCALARS, *float* atol, *float* rtol (Input)
Scalar values that apply to the error estimates of all components of the solution y in the differential equation solver SDASLX. See optional argument IMSL_ATOL_RTOL_ARRAYS if separate tolerances are to be applied to each component of y.
Default: atol and rtol are set to 10^{-3} in single precision and 10^{-5} in double precision.

IMSL_ATOL_RTOL_ARRAYS, *float* atol[], *float* rtol[], (Input)

Componentwise tolerances are used for the computation of solution y in the differential equation solver SDASLX. Arguments atol and rtol are pointers to arrays of length 3*nxgrid to be used for the absolute and relative tolerance and to be applied to each component of the solution, y. See optional argument IMSL_ATOL_RTOL_SCALARS if scalar values of absolute and relative tolerances are to be applied to all components. Default: All elements of atol and rtol are set to 10^{-3} in single precision and 10^{-5} in double precision.

IMSL_NDEGREE, *int* ndeg (Input)

The degree used for the Gauss-Legendre formulas for constructing the finite element matrices. It is required that ndeg ≥ 6.
Default: ndeg = 6.

IMSL_TDEPEND, *int* tdepend[] (Output)

Vector of length 7 indicating time dependence of the coefficients, boundary conditions and function ϕ in the Feynman-Kac equation. If tdepend[i] = 0 then argument i is not time dependent, if tdepend[i]=1 then argument i is time dependent.

i	time dependence of variable
0	σ'
1	σ
2	μ
3	κ
4	Left boundary conditions
5	Right- boundary conditions
6	ϕ

i	time dependence of variable

IMSL_MAX_STEP, *float* max_stepsize (Input)
This is the maximum step size the integrator may take.
Default: max_stepsize = imsl_f_machine(2), the largest possible machine number.

IMSL_INITIAL_STEPSIZE, *float* init_stepsize (Input)
The starting step size for the integration. Must be less than zero since the integration is internally done from t=0 to t=tgrid[ntgrid-1] in a negative direction.
Default: init_stepsize = $-\varepsilon$, where ε is the machine precision.

IMSL_MAX_NUMBER_STEPS, *int* max_steps (Input)
The maximum number of steps between each output point of the integration.
Default: maxsteps = 500000.

IMSL_STEP_CONTROL, *int* step_control (Input)
Indicates which step control algorithm is used for the integration. If step_control = 0, then the step control method of Söderlind is used. If step_control = 1, then the method used by the original Petzold code SASSL is used.
Default: step_control = 0.

IMSL_MAX_BDF_ORDER, *int* max_bdf_order (Input)
Maximum order of the backward differentiation formulas used in the integrator. It is required that $1 \leq$ max_bdf_order ≤ 5.
Default: max_bdf_order = 5.

IMSL_T_BARRIER, *float* t_barrier (Input)
This optional argument controls whether the code should integrate past a special point, t_barrier, and then interpolate to get y and y' at the points in tgrid[]. If this optional argument is present, the integrator assumes the equations either change on the alternate sides of t_barrier or they are undefined there. In this case, the code creeps up to t_barrier in the direction of integration. It is required that t_barrier $\geq$tgrid[ntgrid-1].
Default: t_barrier = tgrid[ntgrid-1].

IMSL_ISTATE, *int* istate[] (Output)
An array of length 7 whose entries flag the state of computation for the matrices and vectors required in the integration. For each entry, a zero indicates that no computation has been done or there is a time dependence. A one indicates that the entry has been computed and there is no time dependence.
The istate entries are as follows:

i	Entry in istate
0	Mass Matrix, M
1	Stiffness matrix, N
2	Bending matrix, R
3	Weighted mass, K
4	Left boundary conditions
5	Right- boundary conditions
6	Forcing term

Default: `istate[i]` = 0 for `i` = 0,…,6.

`IMSL_EVALS`, *int* `nval[]` (Output)
Array of length 3 summarizing the number of evaluations required during the integration.

i	nval[i
0	Number of residual function evaluations of the DAE used in the model.
1	Number of factorizations of the differential matrix associated with solving the DAE.
2	Number of linear system solve steps using the differential matrix.

Description

The generalized Feynman-Kac differential equation has the form

$$f_t + \mu(x,t)\, f_x + \frac{\sigma^2(x,t)}{2} f_{xx} - \kappa(x,t) f = \phi(f, x, t),$$

where the initial data satisfies $f(x,T) = p(x)$. The derivatives are

$$f_t = \frac{\partial f}{\partial t}, \text{ etc}$$

The function `imsl_f_feynman_kac` uses a finite element Galerkin method over the rectangle

$$\left[x_{\min}, x_{\max}\right] \times \left[\overline{T}, T\right]$$

in (x,t) to compute the approximate solution. The interval $\left[x_{\min}, x_{\max}\right]$ is decomposed with a grid

$$\left(x_{\min} =\right) x_1 < x_2 < \ldots < x_m \left(= x_{\max}\right).$$

On each subinterval the solution is represented by

$$f(x,t) = f_i b_0(z) + f_{i+1} b_0(1-z) + h_i f_i' b_1(z) - h_i f_{i+1}' b_1(1-z) + h_i^2 f_i'' b_2(z) + h_i^2 f_{i+1}'' b_2(1-z).$$

The values

$$f_i,\ f_{i,}' f_i'',\ f_{i+1},\ f_{i+1,}' f_{i+1}''$$

are time-dependent coefficients associated with each interval. The basis functions b_0, b_1, b_2 are given for

$$x \in \left[x_i, x_{i+1}\right],\ h_i = x_{i+1} - x_i,\ z = (x - x_i)/h_i,\ z \in [0,1],$$

by

$$b_0(z) = -6z^5 + 15z^4 - 10z^3 + 1 = (1-z)^3\left(6z^2 + 3z + 1\right),$$

$$b_1(z) = -3z^5 + 8z^4 - 6z^3 + z = (1-z)^3 z(3z+1),$$

$$b_2(z) = \frac{1}{2}\left(-z^5 + 3z^4 - 3z^3 + z^2\right) = \frac{1}{2}(1-z)^3 z^2.$$

The Galerkin principle is then applied. Using the provided initial and boundary conditions leads to an Index 1 differential-algebraic equation for the time-dependent coefficients

$$f_i,\ f_{i,}' f_i'',\ f_{i+1},\ f_{i+1,}' f_{i+1}''[= y_i, y_{i+1}, y_{i+2}, \ldots], i = 1, \ldots, m-1.$$

This system is integrated using the variable order, variable step algorithm `DDASLX/SDASLX` noted in Hanson and Krogh, R. (2008) *Solving Constrained Differential-Algebraic Systems Using Projections*. Solution values and their time derivatives are returned at a grid preceding time *T*, expressed in units of time remaining. For further details of deriving and solving the system see Hanson, R. (2008) *Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements*. To evaluate *f* or its partial derivatives at any time point in the grid, use the function `imsl_f_feynman_kac_evaluate`.

Examples

Example 1

The value of the American Option on a Vanilla Put can be no smaller than its European counterpart. That is due to the American Option providing the opportunity to exercise at any time prior to expiration. This example compares this difference – or premium value of the American Option – at two time values using the Black-Scholes model. The example is based on Wilmott et al. (1996, p. 176), and uses the non-linear forcing or weighting term described in Hanson, R. (2008), "*Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements*", for evaluating the price of the American Option. The coefficients, payoff, boundary conditions and forcing term for American and European options are defined by the user functions `fkcfiv_put`, `fkbcp_put` and `fkforce_put`, respectively. One breakpoint is set exactly at the strike price.

The sets of parameters in the computation are:

1. Strike price $K = \{10.0\}$
2. Volatility $\sigma = \{0.4\}$
3. Times until expiration $= \{1/4, 1/2\}$
4. Interest rate $r = 0.1$
5. $x_{\min} = 0.0, \quad x_{\max} = 30.0$
6. $nx = 61, \quad n = 3 \times nx = 183$

The payoff function is the "vanilla option", $p(x) = \max(K - x, 0)$.

Link to example source (feynman_kac_ex1.c)

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define  max(A,B)   ((A) >= (B) ? (A) : (B))
#define NXGRID 61
#define NTGRID 2
#define NV 9

void fkcfiv_put(float, float, int *, float *);
```

```
void fkbcp_put(int, float, int *, float[]);
void fkforce_put(int, int, int, float[], float, float, float[],
               float[], float[], float[],float[], void *);

int main()
{
  /* Compute American Option Premium for Vanilla Put */

  /* The strike price */
  float KS = 10.0;

  /* The sigma value */
  float sigma = 0.4;

  /* Time values for the options */
  int nt = 2;
  float time[] = { 0.25, 0.5};

  /* Values of the underlying where evaluations are made */
  float xs[] = { 0.0, 2.0, 4.0, 6.0, 8.0, 10.0,
                 12.0, 14.0, 16.0 };

  /* Value of the interest rate */
  float r = 0.1;

  /* Values of the min and max underlying values modeled */
  float x_min=0.0, x_max=30.0;

  /* Define parameters for the integration step. */
  int nint=NXGRID-1, n=3*NXGRID;

  float xgrid[NXGRID];
  float ye[(NTGRID+1)*3*NXGRID],yeprime[(NTGRID+1)*3*NXGRID];
  float ya[(NTGRID+1)*3*NXGRID], yaprime[(NTGRID+1)*3*NXGRID];
  float fe[NTGRID*NV], fa[NTGRID*NV];
  float dx;

  int i;
  int nlbcd = 2, nrbcd = 3;

  float atol = 0.2e-2;

  /* Array for user-defined data */
  float usr_data[3];

  /* Define an equally-spaced grid of points for the
     underlying price */
  dx = (x_max-x_min)/((float) nint);
  xgrid[0]=x_min;
  xgrid[NXGRID-1]=x_max;
  for (i=2; i<=NXGRID-1; i++) xgrid[i-1]=xgrid[i-2]+dx;

  usr_data[0] = KS;
  usr_data[1] = r;
  usr_data[2] = atol;

  imsl_f_feynman_kac(NXGRID, nt, nlbcd, nrbcd, xgrid, time,
```

```
                    fkcfiv_put, fkbcp_put, ye, yeprime,
                    IMSL_ATOL_RTOL_SCALARS, 0.5e-2, 0.5e-2, 0);

  imsl_f_feynman_kac(NXGRID, nt, nlbcd, nrbcd, xgrid, time,
                    fkcfiv_put, fkbcp_put, ya, yaprime,
                    IMSL_FCN_FORCE_W_DATA, fkforce_put, usr_data,
                    IMSL_ATOL_RTOL_SCALARS, 0.5e-2, 0.5e-2, 0);

  /* Evaluate solutions at vector of points XS(:), at each
     time value prior to expiration. */

  for (i=0; i<nt; i++)
  {
     imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &ye[(i+1)*n],
                                  IMSL_RETURN_USER, &fe[i*NV], 0);
     imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &ya[(i+1)*n],
                                  IMSL_RETURN_USER, &fa[i*NV], 0);
  }

  printf("\nAmerican Option Premium for Vanilla Put, "
         "3 and 6 Months Prior to Expiry\n");
  printf("%7sNumber of equally spaced spline knots:%4d\n", " ",
         NXGRID);
  printf("%7sNumber of unknowns:%4d\n", " ", n);
  printf("%7sStrike=%6.2f, sigma=%5.2f, Interest Rate=%5.2f\n\n",
         " ",KS,sigma,r);
  printf("%7s%10s%20s%20s\n", " ","Underlying","European","American");
  for (i=0; i<NV; i++)
     printf("%7s%10.4f%10.4f%10.4f%10.4f%10.4f\n", " ", xs[i], fe[i],
            fe[i+NV], fa[i], fa[i+NV]);
}


/* These routines define the coefficients, payoff, boundary conditions
   and forcing term for American and European Options.  */

void fkcfiv_put(float x, float tx, int *iflag, float *value)
{
   /* The sigma value */
   float sigma=0.4;

   /* Value of the interest rate and continuous dividend */
   float strike_price=10.0, interest_rate=0.1, dividend=0.0;

   float zero=0.0;

   switch (*iflag)
   {
      case 0:
        /* The payoff function */
        *value = max(strike_price - x, zero);
      break;

      case -1:
        /* The coefficient derivative d sigma/ dx */
        *value = sigma;
      break;
```

```
      case 1:
        /* The coefficient sigma(x) */
        *value = sigma*x;
      break;

      case 2:
        /* The coefficient mu(x) */
        *value = (interest_rate - dividend) * x;
      break;

      case 3:
        /* The coefficient kappa(x) */
        *value = interest_rate;
      break;
   }

   /* Note that there is no time dependence */
   *iflag = 0;
   return;
}


void fkbcp_put(int nbc, float tx, int *iflag, float val[])
{
   switch (*iflag)
   {
      case 1:
        val[0] =  0.0; val[1] = 1.0; val[2] = 0.0;
        val[3] = -1.0; val[4] = 0.0; val[5] = 0.0;
        val[6] =  1.0; val[7] = 0.0;
      break;

      case 2:
        val[0] = 1.0; val[1]  = 0.0; val[2]  = 0.0;
        val[3] = 0.0; val[4]  = 0.0; val[5]  = 1.0;
        val[6] = 0.0; val[7]  = 0.0; val[8]  = 0.0;
        val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
      break;
   }
   /* Note no time dependence */
   *iflag = 0;
   return;
}


void fkforce_put(int interval, int ndeg, int nxgrid,
                 float y[], float time, float width, float xlocal[],
                 float qw[], float u[], float phi[],float dphi[],
                 void *data_ptr)
{
   int i, j, k, l;
   const int local=6;

   float yl[6], bf[6];
   float value, strike_price, interest_rate, zero=0.0, one=1.0;
   float rt, mu;
```

```
float *data = NULL;

data = (float *) data_ptr;

for (i=0; i<local; i++)
{
   yl[i] = y[3*interval-3+i];
   phi[i] = zero;
}

strike_price = data[0];
interest_rate = data[1];
value = data[2];

mu=2.0;

/* This is the local definition of the forcing term */
for (j=1; j<=local; j++)
  for (l=1; l<=ndeg; l++)
  {
     bf[0] = u[(l-1)];
     bf[1] = u[(l-1)+ndeg];
     bf[2] = u[(l-1)+2*ndeg];
     bf[3] = u[(l-1)+6*ndeg];
     bf[4] = u[(l-1)+7*ndeg];
     bf[5] = u[(l-1)+8*ndeg];

     rt = 0.0;
     for (k=0; k<local; k++)
        rt += yl[k]*bf[k];

     rt = value/(rt + value - (strike_price-xlocal[l-1]));
     phi[j-1] += qw[l-1] * bf[j-1] * pow(rt,mu);
  }

for (i=0; i<local; i++)
  phi[i] = -phi[i]*width*interest_rate*strike_price;

/* This is the local derivative matrix for the forcing term */
for (i=0; i<local*local; i++)
  dphi[i] = zero;

for (j=1; j<=local; j++)
  for (i=1; i<=local; i++)
    for (l=1; l<=ndeg; l++)
    {
       bf[0] = u[(l-1)];
       bf[1] = u[(l-1)+ndeg];
       bf[2] = u[(l-1)+2*ndeg];
       bf[3] = u[(l-1)+6*ndeg];
       bf[4] = u[(l-1)+7*ndeg];
       bf[5] = u[(l-1)+8*ndeg];

       rt = 0.0;
       for (k=0; k<local; k++)
         rt += yl[k]*bf[k];
```

```
            rt = one/(rt + value-(strike_price-xlocal[l-1]));
            dphi[i-1+(j-1)*local] += qw[l-1] * bf[i-1] * bf[j-1]
                                                  * pow(rt, mu+1.0);
         }

      for (i=0; i<local*local; i++)
         dphi[i] = mu * dphi[i] * width * pow(value, mu) *
                               interest_rate * strike_price;

      return;
}
```

Output

```
American Option Premium for Vanilla Put, 3 and 6 Months Prior to Expiry
       Number of equally spaced spline knots:  61
       Number of unknowns: 183
       Strike= 10.00, sigma= 0.40, Interest Rate= 0.10

       Underlying            European            American
           0.0000    9.7534    9.5136   10.0000   10.0000
           2.0000    7.7536    7.5138    8.0000    8.0000
           4.0000    5.7537    5.5156    6.0000    6.0000
           6.0000    3.7615    3.5683    4.0000    4.0000
           8.0000    1.9070    1.9168    2.0170    2.0865
          10.0000    0.6529    0.8548    0.6771    0.9030
          12.0000    0.1632    0.3371    0.1680    0.3519
          14.0000    0.0372    0.1270    0.0373    0.1321
          16.0000    0.0088    0.0483    0.0084    0.0501
```

Example 2

In Beckers (1980) there is a model for a Stochastic Differential Equation of option pricing. The idea is a " constant elasticity of variance diffusion (or CEV) class"

$$dS = \mu S dt + \sigma S^{\alpha/2} dW, \quad 0 \le \alpha < 2$$

The Black-Scholes model is the limiting case $\alpha \to 2$. A numerical solution of this diffusion model yields the price of a call option. Various values of the strike price K, time values, σ and power coefficient α are used to evaluate the option price at values of the underlying price. The sets of parameters in the computation are:

1. power $\alpha = \{2.0, 1.0, 0.0\}$
2. strike price $K = \{15.0, 20.0, 25.0\}$
3. volatility $\sigma = \{0.2, 0.3, 0.4\}$
4. times until expiration $= \{1/12, 4/12, 7/12\}$
5. underlying prices $= \{19.0, 20.0, 21.0\}$
6. interest rate $r = 0.05$

7. $x_{\min} = 0,\ x_{\max} = 60$

8. $nx = 121, \quad n = 3 \times nx = 363$

With this model the Feynman-Kac differential equation is defined by identifying:

$$x: \quad S$$

$$\sigma(x,t): \quad \sigma x^{\alpha/2}; \quad \frac{\partial \sigma}{\partial x} = \frac{a\sigma}{2} x^{\alpha/2-1}$$

$$\mu(x,t): \quad rx$$

$$\kappa(x,t): \quad r$$

$$\phi(f,x,t) \equiv 0$$

The payoff function is the "vanilla option", $p(x) = \max(x - K, 0)$.

Link to example source (feynman_kac_ex2.c)

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define  max(A,B)   ((A) >= (B) ? (A) : (B))
#define NXGRID 121
#define NTGRID 3
#define NV 3

void fcn_fkcfiv(float, float, int *, float *, void *);
void fcn_fkbcp(int, float, int *, float[], void *);

int main()
{
  /* Compute Constant Elasticity of Variance Model for Vanilla Call */

  /* The set of strike prices */
  float KS[] = { 15.0, 20.0, 25.0};

  /* The set of sigma values */
  float sigma[] = { 0.2, 0.3, 0.4};

  /* The set of model diffusion powers */
  float alpha[] = { 2.0, 1.0, 0.0};

  /* Time values for the options */
  int nt = 3;
  float time[] = { 1.0/12.0, 4.0/12.0, 7.0/12.0 };

  /* Values of the underlying where evaluations are made */
```

```
float xs[] = { 19.0, 20.0, 21.0};

/* Value of the interest rate and continuous dividend */
float r=0.05, dividend=0.0;

/* Values of the min and max underlying values modeled */
float x_min=0.0, x_max=60.0;

/* Define parameters for the integration step. */
int nint = NXGRID-1, n=3*NXGRID;
float xgrid[NXGRID], y[(NTGRID+1)*3*NXGRID];
float yprime[(NTGRID+1)*3*NXGRID], f[NTGRID*NV];
float dx;

/* Number of left/right boundary conditions */
int nlbcd = 3, nrbcd = 3;

float usr_data[6];

int i,i1,i2,i3,j;

/* Define equally-spaced grid of points for the underlying price */

dx = (x_max-x_min)/((float) nint);
xgrid[0] = x_min;
xgrid[NXGRID-1] = x_max;

for (i=2; i<=NXGRID-1; i++)
     xgrid[i-1] = xgrid[i-2]+dx;

printf("%2sConstant Elasticity of Variance Model for Vanilla Call\n",
       " ");
printf("%7sInterest Rate:%7.3f\tContinuous Dividend:%7.3f\n",
       " ", r, dividend);
printf("%7sMinimum and Maximum Prices of Underlying:%7.2f%7.2f\n",
          " ",x_min, x_max);
printf("%7sNumber of equally spaced spline knots:%4d \n", " ",
       NXGRID-1);
printf("%7sNumber of unknowns:%4d\n\n", " ",n);
printf("%7sTime in Years Prior to Expiration:%7.4f%7.4f%7.4f\n",
       " ",time[0], time[1], time[2]);
printf("%7sOption valued at Underlying Prices:%7.2f%7.2f%7.2f\n\n",
       " ", xs[0], xs[1], xs[2]);

for (i1=1; i1<=3; i1++)        /* Loop over power */
  for (i2=1; i2<=3; i2++)      /* Loop over volatility */
    for (i3=1; i3<=3; i3++)   /* Loop over strike price  */
    {
       /* Pass data through into evaluation routines. */
       usr_data[0] = KS[i3-1];
       usr_data[1] = x_max;
       usr_data[2] = sigma[i2-1];
       usr_data[3] = alpha[i1-1];
       usr_data[4] = r;
       usr_data[5] = dividend;

       imsl_f_feynman_kac(NXGRID, nt, nlbcd, nrbcd, xgrid, time,
```

```
                        NULL, NULL, y, yprime,
                        IMSL_FCN_FKCFIV_W_DATA, fcn_fkcfiv, usr_data,
                        IMSL_FCN_FKBCP_W_DATA, fcn_fkbcp, usr_data, 0);

         /* Evaluate solution at vector of points xs, at each time
            Value prior to expiration. */
         for (i=0; i<NTGRID; i++)
            imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs,
                         &y[(i+1)*n], IMSL_RETURN_USER, &f[i*NV], 0);

         printf("%2sStrike=%5.2f, Sigma=%5.2f, Alpha=%5.2f\n",
                         " ", KS[i3-1], sigma[i2-1], alpha[i1-1]);

         for (i=0; i<NV; i++)
         {
           printf("%23sCall Option Values%2s", " ", " ");
           for (j=0; j<nt; j++) printf("%7.4f  ", f[j*NV+i]);
                        printf("\n");
         }
         printf("\n");
      }
}

void fcn_fkcfiv(float x, float tx, int *iflag, float *value,
                void *data_ptr)
{
  float sigma, strike_price, interest_rate;
  float alpha, dividend, zero=0.0, half=0.5;
  float *data = NULL;

  data = (float *)data_ptr;

  strike_price = data[0];
  sigma = data[2];
  alpha = data[3];
  interest_rate = data[4];
  dividend = data[5];

  switch (*iflag)
  {
    case 0:
      /* The payoff function */
      *value = max(x - strike_price, zero);
    break;

    case -1:
      /* The coefficient derivative d sigma/ dx */
      *value = half * alpha * sigma * pow(x, alpha*half-1.0);
    break;

    case 1:
      /* The coefficient sigma(x)  */
      *value = sigma * pow(x, alpha*half);
    break;

    case 2:
      /* The coefficient mu(x) */
```

```
      *value = (interest_rate - dividend) * x;
    break;

    case 3:
      /* The coefficient kappa(x) */
      *value = interest_rate;
    break;
  }

  data = NULL;

  /* Note that there is no time dependence */
  *iflag = 0;

  return;
}


void fcn_fkbcp(int nbc, float tx, int *iflag, float val[],
               void *data_ptr)
{
  float x_max, df, interest_rate, strike_price;
  float *data = NULL;

  data = (float *)data_ptr;

  strike_price = data[0];
  x_max = data[1];
  interest_rate = data[4];

  switch (*iflag)
  {
    case 1:
      val[0] = 1.0; val[1]  = 0.0; val[2]  = 0.0;
      val[3] = 0.0; val[4]  = 0.0; val[5]  = 1.0;
      val[6] = 0.0; val[7]  = 0.0; val[8]  = 0.0;
      val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;

      /*  Note no time dependence at left end  */
      *iflag = 0;
    break;

    case 2:
      df = exp(interest_rate*tx);
      val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
      val[3] = x_max - df*strike_price; val[4] = 0.0;
      val[5] = 1.0; val[6] = 0.0; val[7] = 1.0;
      val[8] = 0.0; val[9] = 0.0; val[10] = 1.0;
      val[11] = 0.0;
    break;
  }

  data = NULL;
  return;
}
```

Output

```
Constant Elasticity of Variance Model for Vanilla Call
     Interest Rate:  0.050     Continuous Dividend:  0.000
     Minimum and Maximum Prices of Underlying:   0.00  60.00
     Number of equally spaced spline knots: 120
     Number of unknowns: 363

     Time in Years Prior to Expiration: 0.0833 0.3333 0.5833
     Option valued at Underlying Prices:  19.00  20.00  21.00

Strike=15.00, Sigma= 0.20, Alpha= 2.00
                     Call Option Values   4.0624   4.2577   4.4730
                     Call Option Values   5.0624   5.2507   5.4491
                     Call Option Values   6.0624   6.2487   6.4386

Strike=20.00, Sigma= 0.20, Alpha= 2.00
                     Call Option Values   0.1310   0.5956   0.9699
                     Call Option Values   0.5028   1.0889   1.5100
                     Call Option Values   1.1979   1.7485   2.1751

Strike=25.00, Sigma= 0.20, Alpha= 2.00
                     Call Option Values   0.0000   0.0113   0.0742
                     Call Option Values   0.0000   0.0372   0.1614
                     Call Option Values   0.0007   0.1025   0.3132

Strike=15.00, Sigma= 0.30, Alpha= 2.00
                     Call Option Values   4.0637   4.3399   4.6623
                     Call Option Values   5.0626   5.2945   5.5788
                     Call Option Values   6.0624   6.2709   6.5241

Strike=20.00, Sigma= 0.30, Alpha= 2.00
                     Call Option Values   0.3103   1.0274   1.5500
                     Call Option Values   0.7315   1.5422   2.1024
                     Call Option Values   1.3758   2.1689   2.7385

Strike=25.00, Sigma= 0.30, Alpha= 2.00
                     Call Option Values   0.0006   0.1112   0.3547
                     Call Option Values   0.0038   0.2170   0.5552
                     Call Option Values   0.0184   0.3857   0.8225

Strike=15.00, Sigma= 0.40, Alpha= 2.00
                     Call Option Values   4.0759   4.5136   4.9673
                     Call Option Values   5.0664   5.4199   5.8321
                     Call Option Values   6.0635   6.3577   6.7294

Strike=20.00, Sigma= 0.40, Alpha= 2.00
                     Call Option Values   0.5116   1.4645   2.1286
                     Call Option Values   0.9623   1.9957   2.6944
                     Call Option Values   1.5815   2.6110   3.3230

Strike=25.00, Sigma= 0.40, Alpha= 2.00
                     Call Option Values   0.0083   0.3288   0.7790
                     Call Option Values   0.0285   0.5169   1.0657
                     Call Option Values   0.0813   0.7688   1.4103
```

```
Strike=15.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values   4.0624   4.2479   4.4311
                    Call Option Values   5.0624   5.2479   5.4311
                    Call Option Values   6.0624   6.2479   6.4311

Strike=20.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values   0.0000   0.0241   0.1061
                    Call Option Values   0.1498   0.4102   0.6483
                    Call Option Values   1.0832   1.3313   1.5772

Strike=25.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values   0.0000  -0.0000   0.0000
                    Call Option Values   0.0000   0.0000   0.0000
                    Call Option Values  -0.0000   0.0000   0.0000

Strike=15.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values   4.0624   4.2477   4.4310
                    Call Option Values   5.0624   5.2477   5.4310
                    Call Option Values   6.0624   6.2477   6.4310

Strike=20.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values   0.0016   0.0812   0.2214
                    Call Option Values   0.1981   0.4981   0.7535
                    Call Option Values   1.0836   1.3441   1.6018

Strike=25.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values  -0.0000   0.0000   0.0000
                    Call Option Values  -0.0000   0.0000   0.0000
                    Call Option Values  -0.0000   0.0000   0.0005

Strike=15.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values   4.0624   4.2479   4.4312
                    Call Option Values   5.0624   5.2479   5.4312
                    Call Option Values   6.0624   6.2479   6.4312

Strike=20.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values   0.0072   0.1556   0.3445
                    Call Option Values   0.2501   0.5919   0.8720
                    Call Option Values   1.0867   1.3783   1.6577

Strike=25.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values  -0.0000   0.0000   0.0001
                    Call Option Values   0.0000   0.0000   0.0007
                    Call Option Values   0.0000   0.0002   0.0059

Strike=15.00, Sigma= 0.20, Alpha= 0.00
                    Call Option Values   4.0625   4.2479   4.4311
                    Call Option Values   5.0625   5.2479   5.4312
                    Call Option Values   6.0623   6.2479   6.4312

Strike=20.00, Sigma= 0.20, Alpha= 0.00
                    Call Option Values   0.0001   0.0001   0.0002
                    Call Option Values   0.0816   0.3316   0.5748
                    Call Option Values   1.0818   1.3308   1.5748

Strike=25.00, Sigma= 0.20, Alpha= 0.00
                    Call Option Values   0.0000  -0.0000  -0.0000
```

```
                    Call Option Values   0.0000  -0.0000  -0.0000
                    Call Option Values  -0.0000   0.0000  -0.0000

Strike=15.00, Sigma= 0.30, Alpha= 0.00
                    Call Option Values   4.0624   4.2479   4.4311
                    Call Option Values   5.0625   5.2479   5.4311
                    Call Option Values   6.0623   6.2479   6.4311

Strike=20.00, Sigma= 0.30, Alpha= 0.00
                    Call Option Values   0.0000  -0.0000   0.0029
                    Call Option Values   0.0895   0.3326   0.5753
                    Call Option Values   1.0826   1.3306   1.5749

Strike=25.00, Sigma= 0.30, Alpha= 0.00
                    Call Option Values   0.0000  -0.0000  -0.0000
                    Call Option Values   0.0000  -0.0000  -0.0000
                    Call Option Values   0.0000  -0.0000  -0.0000

Strike=15.00, Sigma= 0.40, Alpha= 0.00
                    Call Option Values   4.0624   4.2479   4.4312
                    Call Option Values   5.0624   5.2479   5.4312
                    Call Option Values   6.0624   6.2479   6.4312

Strike=20.00, Sigma= 0.40, Alpha= 0.00
                    Call Option Values  -0.0000   0.0001   0.0111
                    Call Option Values   0.0985   0.3383   0.5781
                    Call Option Values   1.0830   1.3306   1.5749

Strike=25.00, Sigma= 0.40, Alpha= 0.00
                    Call Option Values   0.0000   0.0000  -0.0000
                    Call Option Values   0.0000  -0.0000  -0.0000
                    Call Option Values   0.0000  -0.0000  -0.0000
```

Example 3

This example evaluates the price of a European Option using two payoff strategies: *Cash-or-Nothing* and *Vertical Spread*. In the first case the payoff function is

$$p(x) = \begin{cases} 0, & x \le K \\ B, & x > K \end{cases}.$$

The value B is regarded as the bet on the asset price, see Wilmott et al. (1995, p. 39-40). The second case has the payoff function

$$p(x) = \max(x - K_1) - \max(x - K_2), \quad K_2 > K_1$$

Both problems use the same boundary conditions. Each case requires a separate integration of the Black-Scholes differential equation, but only the payoff function evaluation differs in each case. The sets of parameters in the computation are:

1. Strike and bet prices K_1={10.0}, K_2 = {15.0}, and B = {2.0}
2. Volatility σ = {0.4}.

3. Times until expiration = {1/4, 1/2}.
4. Interest rate $r = 0.1$.
5. $x_{min} = 0$, $x_{max} = 30$.
6. $nx = 61$, $n = 3 \times nx = 183$.

Link to example source (feynman_kac_ex3.c)

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define max(A,B)   ((A) >= (B) ? (A) : (B))
#define NXGRID 61
#define NTGRID 2
#define NV 12

void fkcfiv_call(float, float, int *, float *, void *);
void fkbcp_call(int, float, int *, float[], void *);

int main()
{
  int i;
  /* The strike price */
  float KS1 = 10.0;
  /* The spread value */
  float KS2 = 15.0;
  /* The Bet for the Cash-or-Nothing Call */
  float bet = 2.0;
  /* The sigma value */
  float sigma = 0.4;

  /* Time values for the options */
  int nt = 2;
  float time[] = {0.25, 0.5};

  /* Values of the underlying where evaluations are made */
  float xs[NV];

  /* Value of the interest rate and continuous dividend */
  float r = 0.1, dividend=0.0;

  /* Values of the min and max underlying values modeled */
  float x_min=0.0, x_max=30.0;

  /* Define parameters for the integration step. */
  int nint=NXGRID-1, n=3*NXGRID;

  float xgrid[NXGRID];
  float yb[(NTGRID+1)*3*NXGRID], ybprime[(NTGRID+1)*3*NXGRID];
  float yv[(NTGRID+1)*3*NXGRID], yvprime[(NTGRID+1)*3*NXGRID];
  float fb[NTGRID*NV], fv[NTGRID*NV];
  float dx;

  /* Number of left/right boundary conditions. */
```

```
int nlbcd = 3, nrbcd = 3;

/* Structure for the evaluation routines. */
struct {
     int idope[1];
     float rdope[7];
} usr_data;

/* Define an equally-spaced grid of points for the underlying
   price */
dx = (x_max-x_min)/((float)(nint));
xgrid[0]=x_min;
xgrid[NXGRID-1]=x_max;
for (i=2; i<=NXGRID-1; i++)
   xgrid[i-1] = xgrid[i-2]+dx;

for (i=1; i<=NV; i++)
   xs[i-1] = 2.0+(i-1)*2.0;

usr_data.rdope[0] = KS1;
usr_data.rdope[1] = bet;
usr_data.rdope[2] = KS2;
usr_data.rdope[3] = x_max;
usr_data.rdope[4] = sigma;
usr_data.rdope[5] = r;
usr_data.rdope[6] = dividend;

/* Flag the difference in payoff functions  */
/* 1 for the Bet, 2 for the Vertical Spread */

usr_data.idope[0] = 1;

imsl_f_feynman_kac(NXGRID, NTGRID, nlbcd, nrbcd, xgrid, time,
                   NULL, NULL, yb, ybprime,
                   IMSL_FCN_FKCFIV_W_DATA, fkcfiv_call, &usr_data,
                   IMSL_FCN_FKBCP_W_DATA, fkbcp_call, &usr_data,
                   0);

usr_data.idope[0] = 2;

imsl_f_feynman_kac(NXGRID, NTGRID, nlbcd, nrbcd, xgrid, time,
                   NULL, NULL, yv, yvprime,
                   IMSL_FCN_FKCFIV_W_DATA, fkcfiv_call, &usr_data,
                   IMSL_FCN_FKBCP_W_DATA, fkbcp_call, &usr_data,
                   0);

/* Evaluate solutions at vector of points XS(:), at each time value
   prior to expiration. */

for (i=0; i<NTGRID; i++)
{
   imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &yb[(i+1)*n],
                          IMSL_RETURN_USER, &fb[i*NV], 0);
   imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &yv[(i+1)*n],
                           IMSL_RETURN_USER, &fv[i*NV], 0);
}
```

```
  printf("%2sEuropean Option Value for A Bet\n", " ");
  printf("%3sand a Vertical Spread, 3 and 6 Months Prior to Expiry\n",
         " ");
  printf("%5sNumber of equally spaced spline knots:%4d\n", " ",
         NXGRID);
  printf("%5sNumber of unknowns:%4d\n", " ", n);
  printf("%5sStrike=%5.2f, Sigma=%5.2f, Interest Rate=%5.2f\n",
         " ", KS1, sigma, r);
  printf("%5sBet=%5.2f, Spread Value=%5.2f\n\n", " ", bet, KS2);
  printf("%17s%18s%18s\n", "Underlying", "A Bet", "Vertical Spread");
  for (i=0; i<NV; i++)
    printf("%8s%9.4f%9.4f%9.4f%9.4f%9.4f\n", " ", xs[i], fb[i],
           fb[i+NV], fv[i], fv[i+NV]);
}


/* These routines define the coefficients, payoff, boundary conditions
                and forcing term for American and European Options.  */

void fkcfiv_call(float x, float tx, int *iflag, float *value,
                 void *data_ptr)
{
  float sigma, strike_price, interest_rate;
  float spread, bet, dividend, zero=0.0;
  float *data_real = NULL;
  int *data_int = NULL;

  struct struct_data {
         int idope[1];
         float rdope[7];
  };

  struct struct_data *data = NULL;

  data = data_ptr;

  data_real = data->rdope;
  data_int = data->idope;

  strike_price = data_real[0];
  bet = data_real[1];
  spread = data_real[2];
  sigma = data_real[4];
  interest_rate = data_real[5];
  dividend = data_real[6];

  switch (*iflag)
  {

    case 0:
      /* The payoff function - Use flag passed to decide which  */
      switch (data_int[0])
      {
        case 1:
          /* After reaching the strike price the payoff jumps
             from zero to the bet value.  */
          *value = zero;
```

```
          if (x > strike_price) *value = bet;
        break;
        case 2:
          /* Function is zero up to strike price.
             Then linear between strike price and spread.
             Then has constant value Spread-Strike Price after
             the value Spread. */
          *value = max(x-strike_price, zero)-max(x-spread, zero);
        break;
      }
    break;

    case -1:
      /* The coefficient derivative d sigma/ dx */
      *value = sigma;
    break;

    case 1:
      /* The coefficient sigma(x) */
      *value = sigma*x;
    break;

    case 2:
      /* The coefficient mu(x) */
      *value = (interest_rate - dividend)*x;
    break;

    case 3:
      /* The coefficient kappa(x) */
      *value = interest_rate;
    break;
  }
  /* Note that there is no time dependence */
  *iflag = 0;

  data_real = NULL;
  data_int = NULL;
  data = NULL;

  return;
}

void fkbcp_call(int nbc, float tx, int *iflag, float val[],
                void *data_ptr)
{
   float strike_price, spread, bet, interest_rate, df;
   int *data_int = NULL;
   float *data_real = NULL;

   struct struct_data {
           int idope[1];
           float rdope[7];
   };

   struct struct_data *data = NULL;

   data = data_ptr;
```

```
   data_int = data->idope;
   data_real = data->rdope;
   strike_price = data_real[0];
   bet = data_real[1];
   spread = data_real[2];
   interest_rate = data_real[5];

   switch (*iflag)
   {
      case 1:
        val[0] = 1.0;  val[1] = 0.0;  val[2] = 0.0;
        val[3] = 0.0;  val[4] = 0.0;  val[5] = 1.0;
        val[6] = 0.0;  val[7] = 0.0;  val[8] = 0.0;
        val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
      /* Note no time dependence in case (1) for IFLAG */
        *iflag = 0;
      break;

      case 2:
        /* This is the discount factor using the risk-free
           interest rate */
        df = exp(interest_rate*tx);
        /* Use flag passed to decide on boundary condition */
        switch (data_int[0])
        {
           case 1:
             val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
             val[3] = bet*df;
           break;
           case 2:
             val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
             val[3] = (spread-strike_price)*df;
          break;
        }
        val[4]  = 0.0; val[5]  = 1.0; val[6] = 0.0;
        val[7]  = 0.0; val[8]  = 0.0; val[9] = 0.0;
        val[10] = 1.0; val[11] = 0.0;
      break;
   }

   data_real = NULL;
   data_int = NULL;
   data = NULL;

   return;
}
```

Output

```
  European Option Value for A Bet
   and a Vertical Spread, 3 and 6 Months Prior to Expiry
     Number of equally spaced spline knots:  61
     Number of unknowns: 183
     Strike=10.00, Sigma= 0.40, Interest Rate= 0.10
     Bet= 2.00, Spread Value=15.00
```

```
Underlying             A Bet   Vertical Spread
    2.0000   0.0000   0.0000   0.0000   0.0000
    4.0000   0.0000   0.0014   0.0000   0.0006
    6.0000   0.0110   0.0722   0.0039   0.0446
    8.0000   0.2690   0.4304   0.1479   0.3831
   10.0000   0.9948   0.9781   0.8909   1.1927
   12.0000   1.6095   1.4287   2.1911   2.2274
   14.0000   1.8654   1.6924   3.4255   3.1551
   16.0000   1.9337   1.8177   4.2264   3.8263
   18.0000   1.9476   1.8700   4.6264   4.2492
   20.0000   1.9501   1.8903   4.7911   4.4922
   22.0000   1.9505   1.8979   4.8497   4.6232
   24.0000   1.9506   1.9007   4.8685   4.6909
```

Example 4

This example evaluates the price of a convertible bond. Here, convertibility means that the bond may, at any time of the holder's choosing, be converted to a multiple of the specified asset. Thus a convertible bond with price x returns an amount K at time T *unless* the owner has converted the bond to $\nu x, \nu \geq 1$, units of the asset at some time prior to T. This definition, the differential equation and boundary conditions are given in Chapter 18 of Wilmott et al. (1996). Using a constant interest rate and volatility factor, the parameters and boundary conditions are:

1. Bond face value $K = \{1\}$, conversion factor $\nu = 1.125$
2. Volatility $\sigma = \{0.25\}$
3. Times until expiration $= \{1/2, 1\}$
4. Interest rate $r = 0.1$, dividend $D = 0.02$
5. $x_{\min} = 0, \quad x_{\max} = 4$
6. $nx = 61, \quad n = 3 \times nx = 183$
7. Boundary conditions $f(0,t) = K\exp(-r(T-t)),\ f(x_{\max},t) = \nu x_{\max}$
8. Terminal data $f(x,T) = \max(K, \nu x)$
9. Constraint for bond holder $f(x,t) \geq \nu x$

Note that the error tolerance is set to a pure absolute error of value 10^{-3}. The free boundary constraint $f(x,t) \geq \nu x$ is achieved by use of a non-linear forcing term in the function `fkforce_cbond`. The terminal conditions are provided with the user function `fkinit_cbond`.

Link to example source (feynman_kac_ex4.c)

```
#include <imsl.h>
#include <stdio.h>
#include <math.h>

#define max(A,B)   ((A) >= (B) ? (A) : (B))
#define NXGRID 61
#define NTGRID 2
#define NV 13

void fkcfiv_cbond(float, float, int *, float *, void *);
void fkbcp_cbond(int, float, int *, float[], void *);
void fkforce_cbond(int, int, int, float[], float, float,
                float[], float[], float[], float[], float[], void *);
void fkinit_cbond(int, int, float[], float[], float,
                    float[], float[], float[], float[], void *);

int main()
{
  int i;
  /* Compute value of a Convertible Bond */

  /* The face value */
  float KS = 1.0e0;

  /* The sigma  or volatility value */
  float sigma = 0.25e0;

  /* Time values for the options */
  float time[] = { 0.5, 1.0};

  /* Values of the underlying where evaluation are made */
  float xs[NV];

  /* Value of the interest rate, continuous dividend and factor */
  float r = 0.1, dividend=0.02, factor = 1.125;

  /* Values of the min and max underlying values modeled */
  float x_min = 0.0, x_max = 4.0;

  /* Define parameters for the integration step. */
  int nint = NXGRID-1, n=3*NXGRID;

  float xgrid[NXGRID];
  float y[(NTGRID+1)*3*NXGRID], yprime[(NTGRID+1)*3*NXGRID];
  float f[(NTGRID+1)*NV], dx;

  /* Array for user-defined data */
  float usr_data[8];
  float atol;

  /* Number of left/right boundary conditions. */
  int nlbcd = 3, nrbcd = 3;

  /*
   * Define an equally-spaced grid of points for the
   * underlying price
   */
```

```
  dx = (x_max-x_min)/((float) nint);
  xgrid[0] = x_min;
  xgrid[NXGRID-1] = x_max;

  for (i=2; i<=NXGRID-1; i++) xgrid[i-1] = xgrid[i-2] + dx;
  for (i=1; i<=NV; i++) xs[i-1] = (i-1)*0.25;

  /* Pass the data for evaluation  */
  usr_data[0] = KS;
  usr_data[1] = x_max;
  usr_data[2] = sigma;
  usr_data[3] = r;
  usr_data[4] = dividend;
  usr_data[5] = factor;

  /* Use a pure absolute error tolerance for the integration */
  atol = 1.0e-3;
  usr_data[6] = atol;

  /* Compute value of convertible bond */
  imsl_f_feynman_kac(NXGRID, NTGRID, nlbcd, nrbcd, xgrid, time,
            NULL, NULL, y, yprime,
            IMSL_FCN_FKCFIV_W_DATA, fkcfiv_cbond, usr_data,
            IMSL_FCN_FKBCP_W_DATA, fkbcp_cbond, usr_data,
            IMSL_FCN_INIT_W_DATA, fkinit_cbond, usr_data,
            IMSL_FCN_FORCE_W_DATA, fkforce_cbond, usr_data,
            IMSL_ATOL_RTOL_SCALARS, 1.0e-3, 0.0e0,
            0);

  /*
   * Evaluate and display solutions at vector of points XS(:),
   * at each time value prior to expiration.
   */

   for (i=0; i<=NTGRID; i++)
       imsl_f_feynman_kac_evaluate (NV, NXGRID, xgrid, xs, &y[i*n],
                           IMSL_RETURN_USER, &f[i*NV], 0);

   printf("%2sConvertible Bond Value, 0+, 6 and 12 Months Prior "
          "to Expiry\n", " ");
   printf("%5sNumber of equally spaced spline knots:%4d\n", " ",
          NXGRID);
   printf("%5sNumber of unknowns:%4d\n", " ",n);
   printf("%5sStrike=%5.2f, Sigma=%5.2f\n", " ", KS, sigma);
   printf("%5sInterest Rate=%5.2f, Dividend=%5.2f, Factor=%6.3f\n\n",
          " ", r, dividend, factor);
   printf("%15s%18s\n", "Underlying", "Bond Value");

   for (i=0; i<NV; i++)
      printf("%7s%8.4f%8.4f%8.4f%8.4f\n",
         " ", xs[i], f[i], f[i+NV], f[i+2*NV]);
}

/*
 * These routines define the coefficients, payoff, boundary conditions
 * and forcing term.
 */
```

```
void fkcfiv_cbond(float x, float tx, int *iflag, float *value,
                          void *data_ptr)
{
   float sigma, strike_price, interest_rate;
   float dividend, factor, zero=0.0;
   float *data = NULL;

   data = (float *) data_ptr;

   strike_price = data[0];
   sigma = data[2];
   interest_rate = data[3];
   dividend = data[4];
   factor = data[5];

   switch(*iflag)
   {
      case 0:
        /*  The payoff function -  */
        *value = max(factor * x, strike_price);
      break;
      case -1:
        /* The coefficient derivative d sigma/ dx */
        *value = sigma;
      break;
      case 1:
        /* The coefficient sigma(x) */
        *value = sigma*x;
      break;
      case 2:
        /* The coefficient mu(x) */
        *value = (interest_rate - dividend) * x;
      break;
      case 3:
        /* The coefficient kappa(x) */
        *value = interest_rate;
      break;
   }
   /* Note that there is no time dependence */
   *iflag = 0;
}

void fkbcp_cbond(int nbc, float tx, int *iflag, float val[],
                         void *data_ptr)
{
   float interest_rate, strike_price, dp, factor, x_max;
   float *data = NULL;

   data = (float *) data_ptr;
   switch (*iflag)
   {
      case 1:
        strike_price = data[0];
        interest_rate = data[3];
        dp = strike_price * exp(tx*interest_rate);
        val[0] = 1.0; val[1]  = 0.0; val[2]  = 0.0;
```

```
        val[3] = dp;  val[4]  = 0.0; val[5]  = 1.0;
        val[6] = 0.0; val[7]  = 0.0; val[8]  = 0.0;
        val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
      break;

      case 2:
        x_max = data[1];
        factor = data[5];
        val[0] = 1.0; val[1] = 0.0; val[2] = 0.0;
        val[3] = factor*x_max; val[4] = 0.0; val[5] = 1.0;
        val[6] = 0.0; val[7] = factor; val[8] = 0.0;
        val[9] = 0.0; val[10] = 1.0; val[11] = 0.0;
        /* Note no time dependence */
        *iflag = 0;
      break;
   }
   return;
}

void fkforce_cbond(int interval, int ndeg, int nxgrid, float y[],
              float time, float width, float xlocal[], float qw[],
              float u[], float phi[], float dphi[], void *data_ptr)
{
   int i, j, k, l;
   const int local=6;

   float yl[6], bf[6];
   float value, strike_price, interest_rate, zero=0.e0;
   float one=1.0e0, rt, mu, factor;

   float *data = NULL;

   data = (float *) data_ptr;

   for (i=0; i<local; i++)
   {
      yl[i] = y[3*interval-3+i];
      phi[i] = zero;
   }

   for (i=0; i<local*local; i++)
      dphi[i] = zero;

   value = data[6];
   strike_price = data[0];
   interest_rate = data[3];
   factor = data[5];

   mu = 2.0;

   /*
    * This is the local definition of the forcing term -
    * It "forces" the constraint f >= factor*x.
    */
   for (j=1; j<=local; j++)
     for (l=1; l<=ndeg; l++)
      {
```

```
        bf[0] = u[(l-1)];
        bf[1] = u[(l-1)+ndeg];
        bf[2] = u[(l-1)+2*ndeg];
        bf[3] = u[(l-1)+6*ndeg];
        bf[4] = u[(l-1)+7*ndeg];
        bf[5] = u[(l-1)+8*ndeg];

        rt = 0.0;
        for (k=0; k<local; k++)
              rt += yl[k]*bf[k];

        rt = value/(rt + value - factor * xlocal[l-1]);
        phi[j-1] += qw[l-1] * bf[j-1] * pow(rt,mu);
     }

     for (i=0; i<local; i++)
       phi[i] = -phi[i]*width*factor*strike_price;

  /*
   * This is the local derivative matrix for the forcing term
   */
   for (j=1; j<=local; j++)
     for (i=1; i<=local; i++)
        for (l=1; l<=ndeg; l++)
        {
           bf[0] = u[(l-1)];
           bf[1] = u[(l-1)+ndeg];
           bf[2] = u[(l-1)+2*ndeg];
           bf[3] = u[(l-1)+6*ndeg];
           bf[4] = u[(l-1)+7*ndeg];
           bf[5] = u[(l-1)+8*ndeg];

           rt = 0.0;
           for (k=0; k<local; k++) rt += yl[k]*bf[k];

           rt = one/(rt + value - factor * xlocal[l-1]);
           dphi[i-1+(j-1)*local] += qw[l-1] * bf[i-1] *
                              bf[j-1] * pow(value*rt, mu) * rt;
         }

   for (i=0; i<local*local; i++)
        dphi[i] = -mu * dphi[i] * width * factor * strike_price;

   return;
}

void fkinit_cbond(int nxgrid, int ntgrid, float xgrid[], float tgrid[],
                  float time, float yprime[], float y[], float atol[],
                  float rtol[], void *data_ptr)
{
  int i;
  float *data = NULL;

  data = (float *) data_ptr;

  if (time == 0.0)
  {
```

```
    /* Set initial data precisely. */
    for (i=1; i<=nxgrid; i++)
    {
       if (xgrid[i-1] * data[5] < data[0])
       {
          y[3*i-3] = data[0];
          y[3*i-2] = 0.0;
          y[3*i-1] = 0.0;
       }
       else
       {
          y[3*i-3] = xgrid[i-1] * data[5];
          y[3*i-2] = data[5];
          y[3*i-1] = 0.0;
       }
    }
  }
  return;
}
```

Output

```
  Convertible Bond Value, 0+, 6 and 12 Months Prior to Expiry
     Number of equally spaced spline knots:  61
     Number of unknowns: 183
     Strike= 1.00, Sigma= 0.25
     Interest Rate= 0.10, Dividend= 0.02, Factor= 1.125

     Underlying        Bond Value
         0.0000  1.0000  0.9512  0.9048
         0.2500  1.0000  0.9512  0.9049
         0.5000  1.0000  0.9513  0.9065
         0.7500  1.0000  0.9737  0.9605
         1.0000  1.1250  1.1416  1.1464
         1.2500  1.4063  1.4117  1.4121
         1.5000  1.6875  1.6922  1.6922
         1.7500  1.9688  1.9731  1.9731
         2.0000  2.2500  2.2540  2.2540
         2.2500  2.5312  2.5349  2.5349
         2.5000  2.8125  2.8160  2.8160
         2.7500  3.0938  3.0970  3.0970
         3.0000  3.3750  3.3781  3.3781
```