User Documentation for DDEM

Hossein ZivariPiran

February 16, 2011

Contents

Chapter 1

Simulation

The simulator module of DDEM is designed to solve the general system of state-dependent neutral delay differential equations (NDDEs),

$$\begin{aligned} y'(t) &= f(t, y(t), y(\alpha_1(t, y; \mathbf{p})), \ldots, y(\alpha_\nu(t, y; \mathbf{p})), \\ &\qquad y'(\alpha_{\nu+1}(t, y; \mathbf{p})), \ldots, y'(\alpha_\omega(t, y; \mathbf{p})); \mathbf{p}), \quad \text{for } t_0(\mathbf{p}) \le t \le t_F, \\ y(t_0) &= y_0(\mathbf{p}), \\ y(t) &= \phi(t; \mathbf{p}), \quad \text{for } t < t_0(\mathbf{p}), \\ y'(t) &= \phi'(t; \mathbf{p}), \quad \text{for } t < t_0(\mathbf{p}), \end{aligned} \tag{1.1}$$

where y, f, and ϕ are $\mathcal{M}$-vector of functions and $\mathbf{p}$ is an $\mathcal{L}$-vector of parameters and $\alpha_k(t, y; \mathbf{p})$, $k = 1, \ldots, \{\nu + \omega\}$ are scalar functions.

Adding parameters to the list of arguments has many advantages, both conceptually and practically; compared to the traditional approach of treating them as constants. These advantages become obvious when doing a sensitivity analysis or solving a parameter estimation problem. However, users doing only simulation often notice that keeping a good programming style with the traditional approach is hard.

Some Special Interesting Cases

- *Retarded* DDEs:

$$y'(t) = f(t, y(t), y(\alpha_1(t, y; \mathbf{p})), \ldots, y(\alpha_\nu(t, y; \mathbf{p}))).$$

- *Constant Delay* DDEs:

$$\alpha_k(t, y; \mathbf{p}) = t - \tau_k, \quad k = 1, \ldots, \{\nu + \omega\}, \tag{1.2}$$

where τ_k is a scalar constant, or a scalar function of only the parameters ($\tau_k = \tau_k(\mathbf{p})$).

- *State-Independent* DDEs:

$$\alpha_k(t, y; \mathbf{p}) = \alpha_k(t; \mathbf{p}), \quad k = 1, \ldots, \{\nu + \omega\}.$$

1.1 Defining the System

The class **ddemSystem** is used for describing the system of DDEs (1.1), including the required functions and constants. After instantiating an object of this class, the following member functions can be called.

```
int create(int nVariables, int nParameters, int nHistorySegments,
           int nEvents)
```

Purpose: This must be called before any other function to allocate the required memories.
Arguments:

nVariables	The dimension of the DDE system ($\mathcal{M}$).	[*Input*]
nParameters	The number of parameters ($\mathcal{L}$).	[*Input*]
nHistorySegments	The number of segments of the history function ϕ; set to 1 for a continuous history.	[*Input*]
nEvents	Number of event functions. Used for hybrid systems; set to 0 otherwise.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int setF(ddemF fFuncs)
```

Purpose: Specifying the right-hand side function (f).
Arguments:

fFuncs	The name of the right-hand side function. See below for its specification.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
void fFuncs (int state, double t, int m, double* y,
             int nu, int omega, double* z,
             int l, double* p,
             double* f,
             ddemUserProvidedJac* userProvidedJac,
             void* comm)
```

Purpose: Computing the right-hand side of the system (f).
Arguments:

state	The state index of the system. Used for hybrid systems; is always 1 otherwise.	[*Input*]
t	t	[*Input*]
m	$\mathcal{M}$	[*Input*]
y	$[\mathbf{m}] \times 1$; $\mathbf{y}[i-1]$ corresponds to y_i.	[*Input*]
nu	ν	[*Input*]
omega	ω	[*Input*]
z	$[(\mathbf{nu}+\mathbf{omega})\times\mathbf{m}]$; $\mathbf{z}[(j-1)*\mathbf{m}+(i-1)]$ corresponds to $y_i(\alpha_j)$ for $1 \le j \le \nu$, and to $y_i'(\alpha_j)$ for $\nu+1 \le j \le \omega$; 1-dimensional array representation is used for efficiency purposes; `Z`$(i-1,j-1)$ may be used as an alternative (see the note below).	[*Input*]
l	$\mathcal{L}$	[*Input*]
p	$[\mathbf{l}] \times 1$; $\mathbf{p}[j-1]$ corresponds to p_j.	[*Input*]
f	$[\mathbf{m}] \times 1$; $\mathbf{f}[i-1]$ corresponds to f_i.	[*output*]
userProvidedJac	*user provided Jacobians*; used for simulation of stiff systems and sensitivity analysis.	[*Input*]
comm	*communication pointer*; see §1.2.	[*Input*]

Note: The macro definition

```
#define Z(I,J)  z[(J) * m + (I)]
```

can be used to simplify the index conversion.

`int setHistory(double* historyConstant)`

Purpose: Specifying a constant history (ϕ = a parameter independent constant).
Arguments:

historyConstant	$[\mathcal{M}] \times 1$; **historyConstant**$[i-1]$ corresponds to ϕ_i.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

`int setHistory(ddemHistoryFunc historyFunc)`

Purpose: Specifying a time/parameter varying history, $\phi(t;\mathbf{p})$.
Arguments:

historyFunc	The name of the history function. See below for its specification.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
void historyFunc(int historySegmentIndex, double t,
                 int l, double* p,
                 int m, double* h,
                 ddemUserProvidedJac* userProvidedJac,
                 void* comm)
```

Purpose: Computing the history, $\phi(t; \mathbf{p})$.
Arguments:

historySegmentIndex	The index of the required segment of the history; a value between 1 and **nHistorySegments**, specified in `create(...)`; with the index 1 corresponding to the segment ending at the starting point t_0.	[*Input*]
t	t	[*Input*]
l	$\mathcal{L}$	[*Input*]
p	$[\mathbf{l}] \times 1$; $\mathbf{p}[j-1]$ corresponds to p_j.	[*Input*]
m	$\mathcal{M}$	[*Input*]
h	$[\mathbf{m}] \times 1$; $\mathbf{h}[i-1]$ corresponds to ϕ_i.	[*output*]
userProvidedJac	*user provided Jacobians*; used for simulation of stiff systems and sensitivity analysis.	[*Input*]
comm	*communication pointer*; see §1.2.	[*Input*]

```
int setHistoryBreaks(double* historyBreaksConstant)
```

Purpose: Specifying constant, parameter independent breaking points between segments of a discontinuous history.
Arguments:

historyBreaksConstant	$[\mathbf{nHistorySegments}-1]\times 1$, where **nHistorySegments** used in `create(...)`; **historyBreaksConstant**$[j-1]$ corresponds to the breaking point between the segments with indices j and $j+1$, where the segment with index 1 is the segment ending at the starting point t_0.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int setHistoryBreaks(ddemHistoryBreaksFunc historyBreaksFunc)
```

Purpose: Specifying parameter varying breaking points between segments of a discontinuous history.
Arguments:

historyBreaksFunc	The name of the function computing the breaking points. See below for its specification.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
void historyBreaksFunc(int historySegmentIndex,
                       int l, double* p,
                       double& historyBreak,
                       ddemUserProvidedJac* userProvidedJac,
                       void* comm)
```

Purpose: Computing breaking points between segments of a discontinuous history.
Arguments:

historySegmentIndex	The index of the segment at the right side of the required breaking point.	[*Input*]
l	$\mathcal{L}$	[*Input*]
p	[**l**] $\times$ 1; **p**[$j-1$] corresponds to p_j.	[*Input*]
historyBreak	The computed value of the breaking point.	[*Output*]
userProvidedJac	*user provided Jacobians*; used for simulation of stiff systems and sensitivity analysis.	[*Input*]
comm	*communication pointer*; see §1.2.	[*Input*]

```
int setT0(double t0Constant)
```

Purpose: Specifying a constant, parameter independent starting point t_0.
Arguments:

t0Constant	The starting point t_0.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int setT0(ddemT0Func t0Func)
```

Purpose: Specifying a parameter varying starting point, $t_0(\mathbf{p})$.
Arguments:

t0Func	The name of the function computing the starting point. See below for its specification.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
void t0Func(int l, double* p,
            double& tStart,
            ddemUserProvidedJac* userProvidedJac,
            void* comm)
```

Purpose: Computing the starting point t_0.
Arguments:

l	$\mathcal{L}$	[*Input*]
p	[**l**] $\times$ 1; **p**[$j-1$] corresponds to p_j.	[*Input*]
tStart	The computed value of the starting point.	[*Output*]
userProvidedJac	*user provided Jacobians*; used for simulation of stiff systems and sensitivity analysis.	[*Input*]
comm	*communication pointer*; see §1.2.	[*Input*]

`int setT0DiscDegree(int` **t0DiscDegree**`)`

Purpose: Specifying the discontinuity degree at the starting point t_0.
Arguments:

t0DiscDegree	The discontinuity degree at the starting point; "**t0DiscDegree** = 0" means a discontinuity in the value; "**t0DiscDegree** = 1" means a continuous value, but a discontinuous first derivative; and so on. "**t0DiscDegree** = 1" is the default; in that case a call to `setT0DiscDegree(...)` is not necessary.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

`int setY0(double*` **y0Constant**`)`

Purpose: Specifying a constant, parameter independent value for y at the starting point t_0. This is only meaningful if "**t0DiscDegree** = 0" in `setT0DiscDegree(...)`. Otherwise, the value is taken from the evaluation of the history function at the starting point, automatically.
Arguments:

y0Constant	[$\mathcal{M}$] $\times$ 1; **y0Constant**[$i-1$] is the suggested starting value for y_i.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

`int setY0(ddemY0Func` **y0Func**`)`

Purpose: Specifying a parameter varying function for the computation of y at the starting point t_0. This is only meaningful if "**t0DiscDegree** = 0" in `setT0DiscDegree(...)`. Otherwise, the y value at the starting point is taken from the evaluation of the history function at that point, automatically.
Arguments:

t0Func	The name of the function computing the starting value. See below for its specification.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
void y0Func(int l, double* p,
            int m, double* y0,
            ddemUserProvidedJac* userProvidedJac,
            void* comm)
```

Purpose: Computing the starting point $y_0(\mathbf{p})$.
Arguments:

l	$\mathcal{L}$	[*Input*]
p	[**l**] $\times 1$; **p**[$j-1$] corresponds to p_j.	[*Input*]
m	$\mathcal{M}$	[*Input*]
y0	[**m**] $\times 1$; **y0**[$i-1$] corresponds to the starting value for y_i.	[*output*]
userProvidedJac	*user provided Jacobians*; used for simulation of stiff systems and sensitivity analysis.	[*Input*]
comm	*communication pointer*; see §1.2.	[*Input*]

```
int setDelayArguments(int nu, int omega, double* delayArgumentsConstant)
```

Purpose: Specifying delays for constant delay DDEs (Eq. 1.2).
Arguments:

nu	ν	[*Input*]
omega	ω	[*Input*]
delayArgumentsConstant	[**nu** + **omega**] $\times 1$; **delayArgumentsConstant**[$k-1$] corresponds to τ_k.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int setDelayArguments(int nu, int omega,
                      ddemDelayArguments delayArgumentsFunc,
                      int stateDependent)
```

Purpose: Specifying the function computing the delay arguments for time/parameter/state-dependent DDEs.
Arguments:

nu	ν	[*Input*]
omega	ω	[*Input*]
delayArgumentsFunc	The name of the function computing the delay arguments. See below for its specification.	[*Input*]
stateDependent	Specifies whether the system is state-dependent (= `DDEM_TRUE`), or state-independent (= `DDEM_FALSE`).	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
void delayArgumentsFunc(double t, int m, double* y,
                        int l, double* p,
                        int nu, int omega, double* alpha,
                        ddemUserProvidedJac* userProvidedJac,
                        void* comm)
```

Purpose: Computing the delay arguments $\alpha_k(t, y; \mathbf{p})$.
Arguments:

t	t	[*Input*]
m	$\mathcal{M}$	[*Input*]
y	[**m**] $\times$ 1; **y**[$i-1$] corresponds to y_i.	[*Input*]
l	$\mathcal{L}$	[*Input*]
p	[**l**] $\times$ 1; **p**[$j-1$] corresponds to p_j.	[*Input*]
nu	ν	[*Input*]
omega	ω	[*Input*]
alpha	[**nu** + **omega**] $\times$ 1; **alpha**[$k-1$] corresponds to α_k.	[*output*]
userProvidedJac	*user provided Jacobians*; used for simulation of stiff systems and sensitivity analysis.	[*Input*]
comm	*communication pointer*; see §1.2.	[*Input*]

```
int checkIntegrity(void)
```

Purpose: Checking to see if all required functions and constants are set by the user in a way that constitute a well-defined system of DDEs.
Return Code: DDEM_YES if well-defined; otherwise an error code giving some information about missing parts. See Chapter 5 for the list of error codes.

1.2 Simulating the System

The class **ddemSimulator** is used for simulating the system of DDEs (1.1). This class has the following constructor,

```
ddemSimulator(ddemIVP2DDE* ivp2dde)
```

Purpose: Constructor.
Arguments:

ivp2dde	An object instantiated from a class derived from the abstract class `ddemIVP2DDE`. See below for its specification.	[*Input*]

`class ddemIVP2DDE`

Purpose: This is an abstract class declaring a general interface for explicit IVP solvers that can be used by our DDE simulator. Interested users can implement a derived class based on an existing IVP solver and use it in their code. `IVP2DDEImprovedCRK` class, provided by us, is based on a particular IVP method developed by Enright and Yan [2]. If using the provided IVP solver, the user should call the following constructor for instantiating an object.

`IVP2DDEImprovedCRK(int` **interp_Flag**`)`

Purpose: Constructor.
Arguments:

interp_Flag	0,1 or 2; used to specify different error control strategies; see [2] for the details.	[*Input*]

After instantiating an object of **ddemSimulator** class, the following member function is invoked by the user.

```
int simulate(ddemSystemInterface* system, double tEnd, double* p,
             double relTolerance, double* absTolerance, double vanishingConst,
             void* commWithUFuncs,
             ddemSimulationSolution* simulationSolution)
```

Purpose: Simulating the system to find an approximate solution satisfying a prescribed tolerance.
Arguments:

system	The system being simulated. `ddemSystemInterface` is an abstract father class for `ddemSystem`, and is used here for generalization purposes; therefore, the user should actually provide an object instantiated from the class `ddemSystem`.	[*Input*]
tEnd	The end point of simulation.	[*Input*]
p	$[\mathcal{L}] \times 1$; The user provided values for the parameters; $\mathbf{p}[j-1]$ corresponds to p_j.	[*Input*]
relTolerance	The relative error tolerance for all components.	[*Input*]
absTolerance	$[\mathcal{M}] \times 1$; A vector of absolute tolerances for different components. **absTolerance**$[i-1]$ specifies the absolute tolerance for y_i.	[*Input*]
vanishingConst	A constant threshold for small delays; a delay is considered vanishing if fallen below this threshold.	[*Input*]
commWithUFuncs	This so-called *communication pointer*, is a head pointer to all common structures needed by the user inside the supplied subroutines defining the system. It will be passed back to the user routines when they are called by the simulator. See [14, Chapter 5] for the details.	[*Input*]
simulationSolution	An structure containing the approximated solution. See §1.2.1 for the details.	[*Output*]

Return Code: `DDEM_SUCCESS` if successfully found an approximate solution; otherwise, an error code giving some information about the possible issue. See Chapter 5 for the list of error codes.

1.2.1 The Solution

The approximated solution is returned in a structure which contains all the information generated and used during the simulation as well as those typically expected by the user. Here we only discuss those components provided for the user.

```
struct  ddemSimulationSolution{
        int nMeshPoints;
        double* meshPoints;

        double** ys;

        int nLambdas;
        int* lambdasDiscDegrees;
        int* lambdasIndices;

        ddemSimulationStatistics* statistics;
        }
```

Elements:

nMeshPoints	The total number of mesh points.
meshPoints	[**nMeshPoints**] $\times$ 1; values of t at the mesh points; **meshPoints**$[0] = t_0$ and **meshPoints**[**nMeshPoints** $- 1] = t_F$.
ys	[**nMeshPoints**] $\times$ $[\mathcal{M}]$; **ys**$[i-1][j-1]$ is the approximated value of y_j at the ith mesh point.
nLambdas	The number of discontinuity points in $[t_0, t_F]$; the discontinuity points are then $\lambda_0, \lambda_1, \ldots, \lambda_{\textbf{nLambdas}}$.
lambdasDiscDegrees	[**nLambdas**] $\times$ 1; **lambdasDiscDegrees**$[i]$ is the degree of discontinuity at λ_i.
lambdasIndices	[**nLambdas**] $\times$ 1; the index of the discontinuity points in the mesh points; $\lambda_i =$ **meshPoints**[**lambdasIndices**$[i]$].
statistics	The statistics of the simulation. See below for its specification.

```
struct  ddemSimulationStatistics{
        int nSteps;
        int nFails;
        int nFEvals;

        int nHistoryEvals;
        int nDelayEvals;
        int nRootFindings;
        int nInterpEvals;

        int nVanishingSteps;
        int nIteraions4VanishingSteps;
        }
```

Elements:

nSteps	The number of successful steps.
nFails	The number of rejected steps.
nFEvals	The number of times f is evaluated.
nHistoryEvals	The number of times the history function is evaluated.
nDelayEvals	The number of times the delay argument function is evaluated.
nRootFindings	The total number of times the root finding is invoked for discontinuity location.
nInterpEvals	The total number of times that the solution at some past point is evaluated using the interpolating polynomial.
nVanishingSteps	The number of steps taken in vanishing mode.
nIteraions4VanishingSteps	The total number of iterations required for convergence during vanishing steps.

1.3 Evaluating the Solution at Off-Mesh Points

The values of the state variables at mesh points are contained in the solution structure, making them directly accessible by the user. If a value at an off-mesh point is required, then the extra (hidden) information returned in the solution structure can be used to compute an approximation. The following member functions of the **ddemSimulator** class that accept a solution structure as an argument can be used for this purpose.

```
int evaluateSolution(double t, double* y,
                     ddemSimulationSolution* simulationSolution,
                     int interpSelect,
                     int betweenLambdasIndex = DDEM_UNKNOWN)
```

Purpose: Computing the state variables at an off-mesh point.
Arguments:

t	t; specifies the off mesh point.	[*Input*]
y	$[\mathcal{M}] \times 1$; $y[i-1]$ will have y_i or y'_i at **t**.	[*Output*]
simulationSolution	The solution structure returned from `simulate(...)`.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
betweenLambdasIndex	The index of the discontinuity points between which the specified point falls; can be used to accelerate the search for the corresponding interpolant. **betweenLambdasIndex** $= 0$ means $\lambda_0 \leq$ **t** $< \lambda_1$, and so on. The default value is `DDEM_UNKNOWN`, meaning that no information is available; in this case this input argument can be omitted.	[*Input*]

Return Code: `DDEM_SUCCESS` if successfully computed an approximate value; otherwise, an error code giving some information about the possible issue. See Chapter 5 for the list of error codes.

```
int evaluateSolution2(double t, double* y,
                      ddemSimulationSolution* simulationSolution,
                      int interpSelect,
                      int betweenMeshPointsIndex)
```

Purpose: Computing the state variables at an off-mesh point when one knows the mesh points between which the specified point falls..

Arguments:

t	t; specifies the off mesh point.	[*Input*]
y	$[\mathcal{M}] \times 1$; **y**$[i-1]$ will have y_i or y'_i at **t**.	[*Output*]
simulationSolution	The solution structure returned from `simulate(...)`.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
betweenMeshPointsIndex	The index of the mesh points between which the specified point falls; can be used to accelerate the search for the corresponding interpolant. **betweenMeshPointsIndex** $= 0$ means **t** lies between the first and second mesh points, and so on.	[*Input*]

Return Code: `DEM_SUCCESS` if successfully computed an approximate value; otherwise, an error code giving some information about the possible issue. See Chapter 5 for the list of error codes.

1.4 Providing an Output in a File

DDEM provides the user with the facility of storing the solution as the values of the state variables at discrete points in an external file. The following functions can be called for this purpose.

```
int ddemOutSolUniform(ddemSimulationSolution* simulationSolution,
                      int outYIndex, int interpSelect, int nOutPoints,
                      string fileName,
                      int precision,
                      ddemSimulator* simulator)
```

Purpose: Providing a table of values at uniform stepping discrete points for a chosen component in a text-like file (suitable for error analysis).
Arguments:

simulationSolution	The solution structure returned from `simulate(...)`.	[*Input*]
outYIndex	The chosen component of y; $0 \leq$ **outYIndex** $< \mathcal{M}$.	[*Input*]
interpSelect	Either DDEM_INTERP that means the value is required, or DDEM_DERIV_INTERP meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **simulationSolution**; usually the same simulator.	[*Input*]

Return Code: DDEM_SUCCESS or DDEM_FAILURE.

```
int ddemOutSolUniformAll(ddemSimulationSolution* simulationSolution,
                         int interpSelect, int nOutPoints,
                         string fileName,
                         int precision,
                         ddemSimulator* simulator)
```

Purpose: Providing a table of values at uniform stepping discrete points for all components in a text-like file (suitable for error analysis).

Arguments:

simulationSolution	The solution structure returned from `simulate(...)`.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **simulationSolution**; usually the same simulator.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int ddemOutSolNonUniform(ddemSimulationSolution* simulationSolution,
                         int outYIndex, int interpSelect, int nOutPoints,
                         string fileName, string fileNameDisc,
                         int precision,
                         ddemSimulator* simulator)
```

Purpose: Providing a table of values at non-uniform discrete points for a chosen component in a text-like file. The same number of points are chosen between each consecutive mesh points. The discontinuities are also taken into account and are stored separately, so that a discontinuous plot can be generated later using this output.

Arguments:

simulationSolution	The solution structure returned from `simulate(...)`.	[*Input*]
outYIndex	The chosen component of y; $0 \leq$ **outYIndex** $< \mathcal{M}$.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
fileNameDisc	The name of the external file in which the information regarding the discontinuities is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **simulationSolution**; usually the same simulator.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int ddemOutSolNonUniformAll(ddemSimulationSolution* simulationSolution,
                            int interpSelect, int nOutPoints,
                            string fileName, string fileNameDisc,
                            int precision,
                            ddemSimulator* simulator)
```

Purpose: Providing a table of values at non-uniform discrete points for all components in a text-like file. The same number of points are chosen between each consecutive mesh points. The discontinuities are also taken into account and are stored separately, so that a discontinuous plot can be generated later using this output.

Arguments:

simulationSolution	The solution structure returned from `simulate(...)`.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
fileNameDisc	The name of the external file in which the information regarding the discontinuities is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **simulationSolution**; usually the same simulator.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

Chapter 2

Sensitivity Analysis

The sensitivity analyzer module of DDEM is designed to find the sensitivity coefficients for the system 1.1. The first-order forward sensitivities are computed using the *variational equations* approach. The original system is coupled with the system describing the evolution of sensitivity coefficients. Hence, an approximation to the state variable and in some cases to its derivative is also computed.

2.1 Defining the System

The system of DDEs should be defined as an object of class **ddemSystem** as described in §1.1. The only additional requirement is the provision of the Jacobians of the functions (f, ϕ, α, ...) as an extra output of the corresponding user provided functions. This output is an object of the class **ddemUserProvidedJac** with the following specifications

```
struct  ddemUserProvidedJac{
        int needed;
        int needed2nd;

        int tIndependent;
        int pIndependent;
        int yIndependent;
        int zIndependent;

        double* tJac;
        double* pJac;
        double* yJac;
        double* zJac;

        int ttIndependent;
        int tpIndependent;

        double* ttJac;
        double* tpJac;
        }
```

This his structure appears in all user provided functions. However, not all the elements are used for every function. Furthermore, the leading dimension is different for each function. In the following we use $\mathcal{NV}$ to specify the dimension of our (in general) vector-valued functions. ($\mathcal{NV} = \mathcal{M}$ for f, ϕ and $y_0(\mathbf{p})$; $\mathcal{NV} = \nu + \omega$ for α; $\mathcal{NV} = 1$ for $t_0(\mathbf{p})$ and the function specifying the breaking points in the history.)
Elements:

needed	Specifies whether the Jacobians **tJac**, **pJac**, **yJac** and **zJac** are required; `DDEM_YES` or `DDEM_NO`.	[*Input*]
needed2nd	Specifies whether the second-order partial derivatives **ttJac** and **tpJac** are required; `DDEM_YES` or `DDEM_NO`.	[*Input*]
tIndependent	`DDEM_NO` by default. Should be changed to `DDEM_YES` if the function is independent of t, in which case **tJac** need not be set.	[*Output*]
pIndependent	`DDEM_NO` by default. Should be changed to `DDEM_YES` if the function is independent of the parameters, in which case **pJac** need not be set.	[*Output*]

yIndependent	DDEM_NO by default. Should be changed to DDEM_YES if the function is independent of the state variable y, in which case **yJac** need not be set.	[*Output*]
zIndependent	DDEM_NO by default. Should be changed to DDEM_YES if the function is independent of delayed state variables $y(\alpha_k)$, in which case **zJac** need not be set.	[*Output*]
tJac	$[\mathcal{NV}] \times 1$; **tJac**$[i-1]$ is $\partial/\partial t$ of the ith component; TJAC$(i-1)$ may be used as an alternative (see the note below).	[*Output*]
pJac	$[\mathcal{NV} \times \mathcal{L}]$; **pJac**$[(i-1) * \mathcal{L} + (j-1)]$ is $\partial/\partial p_j$ of the ith component; 1-dimensional array representation is used for efficiency purposes; PJAC$(i-1,j-1)$ may be used as an alternative (see the note below).	[*Output*]
yJac	$[\mathcal{NV} \times \mathcal{M}]$; **yJac**$[(i-1) * \mathcal{M} + (j-1)]$ is $\partial/\partial y_j$ of the ith component; 1-dimensional array representation is used for efficiency purposes; YJAC$(i-1,j-1)$ may be used as an alternative (see the note below).	[*Output*]
zJac	$[\mathcal{NV} \times (\nu+\omega) \times \mathcal{M}]$; **zJac**$[(i-1) * (\nu+\omega) * \mathcal{M} + (k-1) * \mathcal{M} + (j-1)]$ is $\partial/\partial y_j(\alpha_k)$ of the ith component; 1-dimensional array representation is used for efficiency purposes; ZJAC$(i-1,j-1,k-1)$ may be used as an alternative (see the note below).	[*Output*]
ttIndependent	DDEM_NO by default. Should be changed to DDEM_YES if **tJac** is independent of t, in which case **ttJac** need not be set.	[*Output*]
tpIndependent	DDEM_NO by default. Should be changed to DDEM_YES if **pJac** is independent of t, in which case **tpJac** need not be set.	[*Output*]
ttJac	$[\mathcal{NV}] \times 1$; **ttJac**$[i-1]$ is $\partial^2/\partial t^2$ of the ith component; TTJAC$(i-1)$ may be used as an alternative (see the note below).	[*Output*]
tpJac	$[\mathcal{NV}] \times [\mathcal{L}]$; **pJac**$[(i-1) * \mathcal{L} + (j-1)]$ is $\partial^2/\partial t \partial p_j$ of the ith component; 1-dimensional array representation is used for efficiency purposes; TPJAC$(i-1,j-1)$ may be used as an alternative (see the note below).	[*Output*]

Note: The definitions

```
#define TJAC(I)     userProvidedJac->tJac[(I)]
#define YJAC(I,J)   userProvidedJac->yJac[(I) * m + (J)]
#define ZJAC(I,J,K) userProvidedJac->zJac[(I)*(nu+omega)*m+(K)*m+(J)]
#define TTJAC(I)    userProvidedJac->ttJac[(I)]
#define TPJAC(I,J)  userProvidedJac->tpJac[(J) * l + (I)]
```

can be used to simplify the index conversions, where **userProvidedJac** is the chosen name for the object of class **ddemUserProvidedJac** appearing as an argument in the list of arguments for the user provided functions that define the system. If user wishes to choose a different name, then the chosen name should be used in these macro definitions accordingly.

2.2 Computing the Sensitivities

In DDEM, the class **ddemSensitivityAnalyzer** is used for sensitivity analysis. This class has the following constructor,

```
ddemSensitivityAnalyzer()
```

After instantiating an object of **ddemSensitivityAnalyzer** class, the following member function is invoked by the user.

```
int computeSensitivities(ddemSimulator* simulator,
                         ddemSystem* baseSystem, double tEnd, double* p,
                         double relTolerance, double* absTolerance,
                         double vanishingConst,
                         int* needSensitivity,
                         void* commWithUFuncs,
                         ddemSensitivitySolution* sensitivitySolution)
```

Purpose: Computing the first-order sensitivity coefficients by simulating the coupled system.
Arguments:

simulator	The simulator being used for integrating the coupled DDE.	[*Input*]
baseSystem	The original system of DDEs for which the sensitivities are being computed.	[*Input*]
tEnd	The end point of simulation.	[*Input*]
p	$[\mathcal{L}] \times 1$; The user provided values for the parameters; **p**$[j-1]$ corresponds to p_j.	[*Input*]
relTolerance	The relative error tolerance for all components.	[*Input*]
absTolerance	$[\mathcal{M}] \times 1$; A vector of absolute tolerances for different components. **absTolerance**$[i-1]$ specifies the absolute tolerance for y_i. The absolute errors for the sensitivities are automatically chosen inside the routine.	[*Input*]
vanishingConst	A constant threshold for small delays; a delay is considered vanishing if fallen below this threshold.	[*Input*]
needSensitivity	$[\mathcal{L}] \times 1$; determines the selected parameters for which sensitivities are to be computed; **needSensitivity**$[j-1]$ = `DDEM_YES` means the $\partial y/\partial p_j$ is required; should be set to `DDEM_NO` if not required.	[*Input*]
commWithUFuncs	This so-called *communication pointer*, is a head pointer to all common structures needed by the user inside the supplied subroutines defining the system. It will be passed back to the user routines when they are called by the simulator. See [14, Chapter 5] for the details.	[*Input*]
sensitivitySolution	An structure containing the approximated coupled solution. See §2.2.1 for the details.	[*Output*]

Return Code: `DDEM_SUCCESS` if successfully found an approximate solution; otherwise, an error code giving some information about the possible issue. See Chapter 5 for the list of error codes.

2.2.1 The Sensitivity Solution

The structure **ddemSensitivitySolution** that represents the result of sensitivity analysis is derived from the structure **ddemSimulationSolution** which represents the simulation result. Hence, all the elements of **ddemSimulationSolution** discussed in §**??** are also present in **ddemSensitivitySolution**. Here we list the elements, but only discuss the differences.

```
struct  ddemSensitivitySolution{
        int nMeshPoints;
        double* meshPoints;

        double** ys; /*new interpretation*/

        int nLambdas;
        int* lambdasDiscDegrees;
        int* lambdasIndices;

        ddemSimulationStatistics* statistics;

        double** lambdasJac; /*new*/
        }
```

Elements:

nMeshPoints The total number of mesh points.

meshPoints [**nMeshPoints**] $\times$ 1; values of t at the mesh points; **meshPoints**$[0] = t_0$ and **meshPoints**[**nMeshPoints** $- 1] = t_F$.

ys [**nMeshPoints**] $\times [\mathcal{M} \times (1 + nS + 0/1)]$, where nS is the number of selected parameters and the $0/1$ choice depends on whether the problem is retarded or neutral, respectively; **ys**$[i-1][k * \mathcal{M} + (j-1)]$ is the approximated value of y_j at the ith mesh point for $k = 0$, the approximated value of $\frac{\partial y_j}{\partial p_k^{\text{select}}}$ for $1 \leq k \leq nS$, and the approximated value of y_j' for $k = nS + 1$ if the system is neutral. (p_k^{select} is the kth selected parameter.)

nLambdas The number of discontinuity points in $[t_0, t_F]$; the discontinuity points are then $\lambda_0, \lambda_1, \ldots, \lambda_{\textbf{nLambdas}}$.

lambdasDiscDegrees [**nLambdas**] $\times$ 1; **lambdasDiscDegrees**$[i]$ is the degree of discontinuity at λ_i.

lambdasIndices [**nLambdas**] $\times$ 1; the index of the discontinuity points in the mesh points; $\lambda_i =$ **meshPoints**[**lambdasIndices**$[i]$].

statistics The statistics of the simulation. See below for its specification.

lambdasJac [**nLambdas**] $\times [nS]$, where nS is the number of selected parameters; **lambdasJac**$[i][k-1]$ is $\frac{d\lambda_i}{dp_k^{\text{select}}}$. ($p_k^{\text{select}}$ is the kth selected parameter.)

2.3 Evaluating the Sensitivity Coefficients at Off-Mesh Points

The methods `evaluateSolution(...)` and `evaluateSolution2(...)` of the simulator class **ddemSimulator** can be used for the evaluation of the sensitivity coefficients at off-mesh points. The details are similar as §1.3, except that the input is an object of class **ddemSensitivitySolution** instead of **ddemSimulationSolution**. The return argument **y** changes as below:

y $[\mathcal{M} \times (1 + nS + 0/1)]$, where nS is the number of selected parameters and the $0/1$ choice depends on whether the problem is retarded or neutral, respectively; $\mathbf{y}[k * \mathcal{M} + (j-1)]$ will have y_j at the required point for $k = 0$, $\frac{\partial y_j}{\partial p_k^{\text{select}}}$ for $1 \leq k \leq nS$, and y'_j for $k = nS + 1$ if the system is neutral. (p_k^{select} is the kth selected parameter.) [*Output*]

2.4 Providing an Output in a File

`DDEM` provides routines for storing the sensitivity coefficients. The calling sequences are similar to those of the simulation §1.4. Here we only discuss the differences.

```
int ddemOutSensUniform(ddemSensitivitySolution* sensitivitySolution,
                       int yIndex, int pIndex,
                       int interpSelect, int nOutPoints,
                       string fileName,
                       int precision,
                       ddemSimulator* simulator)
```

Purpose: Providing a table of values at uniform stepping discrete points for a chosen sensitivity coefficient in a text-like file (suitable for error analysis).
Arguments:

sensitivitySolution	The sensitivity solution structure returned from `computeSensitivities(...)`.	[*Input*]
yIndex	The chosen component of y; $0 \leq$ **yIndex** $< \mathcal{M}$.	[*Input*]
pIndex	The chosen parameter of $\mathbf{p}$; $0 \leq$ **pIndex** $< \mathcal{L}$. $\partial y_{\textbf{yIndex}+1}/\partial p_{\textbf{pIndex}+1}$ is chosen for output.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **sensitivitySolution**; usually the same simulator.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int ddemOutSensUniformAll(ddemSensitivitySolution* sensitivitySolution,
                          int interpSelect, int nOutPoints,
                          string fileName,
                          int precision,
                          ddemSimulator* simulator)
```

Purpose: Providing a table of values at uniform stepping discrete points for all components including the state variables and sensitivity coefficients in a text-like file (suitable for error analysis).
Arguments:

sensitivitySolution	The sensitivity solution structure returned from `computeSensitivities(...)`.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **sensitivitySolution**; usually the same simulator.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int ddemOutSensNonUniform(ddemSensitivitySolution* sensitivitySolution,
                          int yIndex, int pIndex,
                          int interpSelect, int nOutPoints,
                          string fileName, string fileNameDisc,
                          int precision,
                          ddemSimulator* simulator)
```

Purpose: Providing a table of values at non-uniform discrete points for a chosen sensitivity coefficient in a text-like file. The same number of points are chosen between each consecutive mesh points. The discontinuities are also taken into account and are stored separately, so that a discontinuous plot can be generated later using this output.
Arguments:

sensitivitySolution	The sensitivity solution structure returned from `computeSensitivities(...)`.	[*Input*]
yIndex	The chosen component of y; $0 \leq$ **yIndex** $< \mathcal{M}$.	[*Input*]
pIndex	The chosen parameter of $\mathbf{p}$; $0 \leq$ **pIndex** $< \mathcal{L}$. $\partial y_{\textbf{yIndex}+1}/\partial p_{\textbf{pIndex}+1}$ is chosen for output.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
fileNameDisc	The name of the external file in which the information regarding the discontinuities is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **sensitivitySolution**; usually the same simulator.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

```
int ddemOutSensNonUniformAll(ddemSensitivitySolution* sensitivitySolution,
                             int interpSelect, int nOutPoints,
                             string fileName, string fileNameDisc,
                             int precision,
                             ddemSimulator* simulator)
```

Purpose: Providing a table of values at non-uniform discrete points for all components including the state variables and sensitivity coefficients in a text-like file. The same number of points are chosen between each consecutive mesh points. The discontinuities are also taken into account and are stored separately, so that a discontinuous plot can be generated later using this output.

Arguments:

sensitivitySolution	The sensitivity solution structure returned from `computeSensitivities(...)`.	[*Input*]
interpSelect	Either `DDEM_INTERP` that means the value is required, or `DDEM_DERIV_INTERP` meaning the derivative is required.	[*Input*]
nOutPoints	The number of output discrete points; specifying the output resolution.	[*Input*]
fileName	The name of the external file in which the output is written.	[*Input*]
fileNameDisc	The name of the external file in which the information regarding the discontinuities is written.	[*Input*]
precision	The number of decimal points used for storing floating point numbers.	[*Input*]
simulator	A simulator of the same class with that used for producing **sensitivitySolution**; usually the same simulator.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

Chapter 3

Parameter Estimation

Given a set of data $\{Y(\gamma_i) \approx y(\gamma_i; \mathbf{p})\}$ corresponding to a parameterized model (1.1) and a set of constraints the parameter estimation module of DDEM finds the best choice for vector $\mathbf{p}$ satisfying the constraints. The objective function being minimized is the squared two-norm,

$$W(\mathbf{p}) = \sum_i [Y(\gamma_i) - y(\gamma_i; \mathbf{p})]^2 . \tag{3.1}$$

The current implementation uses `nag_opt_nlin_lsq(e04unc)` [] from the NAG C Library for solving the constraint least-squares optimization problem.

3.1 Identifying the Problem

The system of DDEs should be defined as an object of class **ddemSystem** as described in §1.1, and the Jacobians of the functions should be provided as discussed in §2.1.

3.1.1 The Input Data

The class **ddemData** should be used for providing the input data. Currently, this class acts like an intermediate medium between the file containing the data and the parameter estimator. This class has the following constructor,

`ddemData()`

After instantiating an object of this class, the following member function should be called.

`int Load(string fileName)`

Purpose: Reading the file that contains the input data and loading the content into a memory structure.
Arguments:

fileName	The name of the external file from which the input is read.	[*Input*]

Return Code: `DDEM_SUCCESS` or `DDEM_FAILURE`.

The Format of the Input File

The input file is a text-like file with the following format. The first line contains the number of observation points γ_i followed by the total number of scalar observations, separated by at least one space. At each observation point at least one of the components of the state variable y should be present. There could be different components for each γ_i.

Starting the second line, each line starts with γ_i, followed by the number of components of y that are available as observations, followed by space separated pairs of "index"-"value" of those components. Where the integer "index" ($1 \leq$ "index" $\leq \mathcal{M}$) specifies the component and "value" is a floating point representation of the value of that components.

3.1.2 The Constraints

The user may wish to put some constraints on the parameter values. These constraints can be classified in three groups: simple bounds, linear constraints, and non-linear constraints. This classification is suggested by many optimization routines for efficiency purposes. The mathematical description is

$$l_S \leq \mathbf{p} \leq u_S, \tag{3.2}$$

$$l_L \leq A\mathbf{p} \leq u_L, \tag{3.3}$$

$$l_N \leq \mathrm{c}(\mathbf{p}) \leq u_N, \tag{3.4}$$

where l_S, u_S specify lower and upper bounds for the parameter, A is an $n_L \times \mathcal{L}$ constant matrix that specifies, along with l_L, u_L, the linear constraints on the parameters and $\mathrm{c}(\mathbf{p})$ is an n_N-vector valued function that specifies the non-linear constraints with l_N, u_N lower and upper bounds.

The class **ddemConstraints** can be used to specify the constraints. This class has the following constructor,

`ddemConstraints()`

After instantiating an object of this class, the following member functions can be called.

`void setSimpleLowerBound(int` **pIndex**`, double` **lowerBound**`)`

Purpose: Putting a simple lower bound on a chosen parameter.
Arguments:

pIndex	The index of the chosen parameter. ($0 \leq$ **pIndex** $\leq \mathcal{L}-1$)	[*Input*]
lowerBound	The value of the bound to be set; **lowerBound** $\leq p_{\textbf{pIndex}+1}$.	[*Input*]

`void setSimpleUpperBound(int` **pIndex**`, double` **upperBound**`)`

Purpose: Putting a simple upper bound on a chosen parameter.
Arguments:

pIndex	The index of the chosen parameter. ($0 \leq$ **pIndex** $\leq \mathcal{L}-1$)	[*Input*]
upperBound	The value of the bound to be set; $p_{\textbf{pIndex}+1} \leq$ **upperBound**.	[*Input*]

`void setSimpleEquality(int` **pIndex**`, double` **fixedValue**`)`

Purpose: Putting an equality constraint on a chosen parameter.
Arguments:

pIndex	The index of the chosen parameter. ($0 \leq$ **pIndex** $\leq \mathcal{L}-1$)	[*Input*]
fixedValue	The fixed value of chosen parameter; $p_{\textbf{pIndex}+1} =$ **fixedValue**.	[*Input*]

`void setLinearConstraintsMatrix(double*` **linearConstraintsMatrix**`);`

Purpose: Specifying the matrix that defines the coefficients of the linear constraints.
Arguments:

linearConstraintsMatrix	$[n_L \times \mathcal{L}]$; **linearConstraintsMatrix**$[(i-1)*\mathcal{L}+(j-1)]$ refers to $A_{i,j}$ of Eq. (3.3).	[*Input*]

`void setLinearConstraintLowerBound(int` **linearConstraintIndex**`, double` **lowerBound**`)`

Purpose: Specifying a lower bound for a chosen linear inequality constraint.
Arguments:

linearConstraintIndex	The index of the linear constraint. (1 $\leq$ **linearConstraintIndex** $\leq n_L$)	[*Input*]
lowerBound	The value of the bound to be set.	[*Input*]

```
void setLinearConstraintUpperBound(int linearConstraintIndex,
                                   double upperBound)
```

Purpose: Specifying an upper bound for a chosen linear inequality constraint.
Arguments:

linearConstraintIndex	The index of the linear constraint. (1 $\leq$ **linearConstraintIndex** $\leq n_L$)	[*Input*]
upperBound	The value of the bound to be set.	[*Input*]

```
void setLinearConstraintEquality(int linearConstraintIndex,
                                 double equalityRightHandSide)
```

Purpose: Specifying a fixed value for the right-hand side of a chosen linear equality constraint.
Arguments:

linearConstraintIndex	The index of the linear constraint. (1 $\leq$ **linearConstraintIndex** $\leq n_L$)	[*Input*]
equalityRightHandSide	The value of the right-hand side.	[*Input*]

```
void setNonlinearConstraints(ddemNonlinearConstraints nonlinearConstraints)
```

Purpose: Specifying the function that defines the the non-linear constraints ($c(\mathbf{p})$ of 3.4).
Arguments:

nonlinearConstraints	The vector-valued non-linear constraint function. See below for its specification.	[*Input*]

```
void nonlinearConstraints (int l, int ncnlin, int needc[], double p[],
                           double conf[], double cjac[], int flag)
```

Purpose: Computing the non-linear constraints $c(\mathbf{p})$.
Arguments: The argument list is desined to be compatible with the type of nonlinear constraints used in `nag_opt_nlin_lsq(e04unc)` [], here we give a brief description. (See the reference for a detailed description.)

l	$\mathcal{L}$	[*Input*]
ncnlin	The number of non-linear constraints n_N.	[*Input*]
needc	[**ncnlin**] $\times$ 1; **needc**$[i-1] > 0$ indicates that the ith component of $c(\mathbf{p})$ and, (if applicable) its corresponding partial derivatives are needed.	[*Input*]
p	[**l**] $\times$ 1; **p**$[j-1]$ corresponds to p_j.	[*Input*]
conf	[**ncnlin**] $\times$ 1; **conf**$[i-1]$ should be set to the computed value of the ith component of $c(\mathbf{p})$, if the ith component is required.	[*output*]
cjac	[**ncnlin** $\times$ **l**]; the Jacobian of the constraint function; **cjac**$[(i-1)*\mathbf{l}+(j-1)$ corresponds to $\partial c_i/\partial p_j$.	[*output*]
flag	if **flag** $= 0$, only **conf** is required; if **flag** $= 2$, both **conf** and **cjac** are required.	[*Input*]

`void setNonlinearConstraintLowerBound(int` **nonlinearConstraintIndex**`, double` **lowerBound**`)`

Purpose: Specifying a lower bound for a chosen non-linear inequality constraint.

Arguments:

nonlinearConstraintIndex	The index of the non-linear constraint. ($1 \leq$ **nonlinearConstraintIndex** $\leq n_N$)	[*Input*]
lowerBound	The value of the bound to be set.	[*Input*]

`void setNonlinearConstraintUpperBound(int` **nonlinearConstraintIndex**`, double` **upperBound**`)`

Purpose: Specifying an upper bound for a chosen non-linear inequality constraint.

Arguments:

nonlinearConstraintIndex	The index of the linear constraint. ($1 \leq$ **nonlinearConstraintIndex** $\leq n_N$)	[*Input*]
upperBound	The value of the bound to be set.	[*Input*]

`void setNonlinearConstraintEquality(int` **nonlinearConstraintIndex**`, double` **equalityRightHandSide**`)`

Purpose: Specifying a fixed value for the right-hand side of a chosen non-linear equality constraint.

Arguments:

nonlinearConstraintIndex	The index of the non-linear constraint. ($1 \leq$ **nonlinearConstraintIndex** $\leq n_L$)	[*Input*]
equalityRightHandSide	The value of the right-hand side.	[*Input*]

3.2 Finding the Parameters

The class **ddemParameterEstimator** is used for parameter estimation. This class has the following constructor,

```
ddemParameterEstimator()
```

After instantiating an object of **ddemParameterEstimator** class, the following member function is invoked by the user.

```
int EstimateParameters(ddemSimulator* simulator,
                       ddemSyatem* system, ddemData* data,
                       ddemConstraints* constraints, double* p,
                       double relTolerance, double* absTolerance,
                       ddemParameterEstimationStatistics* statistics,
                       int peSelect,
                       void* commWithUFuncs = NULL)
```

Purpose: Finding parameter values corresponding to the best-fit for given data and satisfying constraints.
Arguments:

simulator	The simulator being used for integrating systems of DDEs; it is used as the base integrator of sensitivity analyzer (if required).	[*Input*]
system	Identifies the underlying system of DDEs.	[*Input*]
data	The observations loaded in memory.	[*Input*]
constraints	The constraints being imposed on the parameters.	[*Input*]
p	The initial guess for the parameters (on entry). The value of optimized parameters (after return).	[*Input/output*]
relTolerance	The relative error tolerance for all components of underlying DDE; passed to the simulator whenever invoked.	[*Input*]
absTolerance	$[\mathcal{M}] \times 1$; A vector of absolute tolerances for different components. **absTolerance**$[i-1]$ specifies the absolute tolerance for y_i; passed to the simulator whenever invoked.	[*Input*]
statistics	The statistics of the parameter estimation. See below for its specification..	[*Input*]
peSelect	`DDEM_PE_VERY_SIMPLE`: No added constraints, divided differences Jacobians; `DDEM_PE_SIMPLE`: No added constraints, variational equations Jacobians; `DDEM_PE_NEW`: Discontinuity passing prevention by added constraints; `DDEM_PE_NEW_CE`: Discontinuity passing prevention by added constraints, extrapolation used for infeasible parameter values; (see [14, Chapter 5]).	[*Input*]
commWithUFuncs	*communication pointer*; see §1.2.	[*Input*]

Return Code:`DDEM_SUCCESS` if successfully converged an found the best-fit parameters; otherwise, an error code giving some information about the possible issue. See Chapter 5 for the list of error codes.

```
struct  ddemParameterEstimationStatistics: public ddemSimulationStatistics{
        int nReferencedParameterValues;
        double finalObjectiveValue;
        }
```

Elements:

...	All the elements of class `ddemSimulationStatistics`; see §1.2.1.
nReferencedParameterValues	The number of different parameters for which parameter estimator was asked to provide values and/or Jacobians by the optimizer (usually called number of iterations in an iterative optimization scheme).
finalObjectiveValue	The value of the objective function (sum of squares) after final return from the optimizer.

Chapter 4

Test Examples and Drivers

We provide the driver and coded systems for a collection of DDEs which have been studied by us as test examples during the development of our package DDEM.

4.1 Description of Sample Systems

The function/subroutines defining the following systems of DDEs are provided.

System 1 [9]:

$$y' = y(y(t)),$$

for t in $[2, 5.5]$. The history function is

$$y = 0.5, \quad \text{for } t < 2,$$

and

$$y(2) = 1.$$

The C^0 discontinuity of the solution at $\xi_0 = 2$ introduces break points at $\xi_1 = 4$ (C^1) and $\xi_2 = 4 + 2\ln 2 \approx 5.386$ (C^2).

The exact solution to this problem is

$$y(t) = \begin{cases} t/2, & \text{for } \xi_0 \le t \le \xi_1, \\ 2\exp(t/2 - 2), & \text{for } \xi_1 \le t \le \xi_2, \\ 4 - 2\ln(1 + \xi_2 - t) & \text{for } \xi_2 \le t \le 5.5. \end{cases}$$

The parameter is,

$$\mathbf{p} = [y(2)].$$

System 2. A neutral delay logistic Gause-type predator-prey system [5]:

$$y_1'(t) = y_1(t)(1 - y_1(t-\tau) - \rho y_1'(t-\tau)) - \frac{y_2(t)y_1(t)^2}{y_1(t)^2+1},$$
$$y_2'(t) = y_2(t)\left(\frac{y_1(t)^2}{y_1(t)^2+1} - \alpha\right),$$

where $\alpha = 1/10$, $\rho = 29/10$ and $\tau = 21/50$, for t in $[0, 30]$. The history functions are

$$\phi_1(t) = \frac{33}{100} - \frac{1}{10}t,$$
$$\phi_2(t) = \frac{111}{50} + \frac{1}{10}t,$$

for $t \leq 0$. The solution is C^1 discontinuous at the starting point which propagates as C^1 and C^2 discontinuities to $y_1(t)$ and $y_2(t)$, respectively, at $t = n\tau$ for $n \geq 1$.
The exact solution of this problem is unknown.

The parameters are,

$$\mathbf{p} = [\tau,\ \rho,\ \alpha,\ \mathrm{a},\ \mathrm{b},\ \mathrm{c},\ \mathrm{d}],$$

where

$$\phi_1(t) = \mathrm{a} + \mathrm{b}\,t,$$
$$\phi_2(t) = \mathrm{c} + \mathrm{d}\,t.$$

System 3 [8]:

$$y'(t) = \frac{y(t)y(\ln(y(t)))}{t},$$

for t in $[1, 10]$. The history function is

$$\phi(t) = 1, \quad \text{for} \quad t \leq 1.$$

The exact solution to this problem is

$$y(t) = \begin{cases} t, & \text{for } 1 \leq t \leq e, \\ \exp(t/e), & \text{for } e \leq t \leq e^2, \\ \left(\frac{e}{3-\ln(t)}\right)^e, & \text{for } e^2 \leq t \leq e_3, \\ \text{not known}, & \text{for } e_3 < t, \end{cases}$$

where $e_3 = \exp(3 - \exp(1 - e))$.
Derivative jump discontinuities occur at $t = 1$ (C^1), $t = e$ (C^2), $t = e^2$ (C^3) and $t = e_3$ (C^4).

The parameter is,

$$\mathbf{p} = [t_0].$$

System 4 [10]:

$$y'(t) = y(t - t^{-10}),$$

for t in [1, 10]. The history function is

$$\phi(t) = t, \quad \text{for} \quad t \leq 1.$$

The exact solution of this problem is unknown.
This DDE has a vanishing (but non-singular) lag ($\lim_{t\to+\infty} t^{-10} = 0$; however, depending on the precision used, the vanishing behavior will first be recognized at some finite time $t^\star$ and persists for all $t > t^\star$).

System 5 [13]:

$$y'(t) = y(y(t)) + (3 + \mu)t^{(2+\mu)} - t^{(3+\mu)^2},$$

for t in [0, 1]. The initial value is

$$y(0) = 0.$$

The exact solution to this problem is

$$y(t) = t^{(3+\mu)}, \quad \text{for} \quad 0 \leq t \leq 1.$$

This is an *initial value* DDE with no discontinuities.
We use $\mu = 0$ in our experiments. The exact solution is a low degree polynomial and any IVP method should have no trouble with this problem.

The parameter is,

$$\mathbf{p} = [\mu].$$

System 6. The SEIR epidemic model of Genik & van den Driessche [3]:

$$\begin{aligned}
S' &= A - dS(t) - \lambda\tfrac{S(t)I(t)}{N(t)} + \gamma I(t-\tau)e^{-d\tau},\\
E' &= \lambda\tfrac{S(t)I(t)}{N(t)} - \lambda\tfrac{S(t-\omega)I(t-\omega)}{N(t-\omega)}e^{-d\omega} - dE(t),\\
I' &= \lambda\tfrac{S(t-\omega)I(t-\omega)}{N(t-\omega)}e^{-d\omega} - (\gamma + \epsilon + d)I(t),\\
R' &= \gamma I(t) - \gamma I(t-\tau)e^{-d\tau} - dR(t),
\end{aligned}$$

where

$$N(t) = S(t) + E(t) + I(t) + R(t),$$

and $A = 0.33$, $d = 0.006$, $\lambda = 0.308$, $\gamma = 0.04$, $\epsilon = 0.06$, $\tau = 42$, $\omega = 0.15$, for t in $[0, 350]$. The history functions are

$$\begin{aligned} S &= 15, \\ E &= 0, \\ I &= 2, \\ R &= 3, \end{aligned}$$

for $t \leq 0$.
The exact solution of this problem is unknown.

The parameters are,

$$\mathbf{p} = [A,\ d,\ \lambda,\ \gamma,\ \epsilon,\ \tau,\ \omega].$$

System 7. Model of hematopoiesis (Mahaffy, Bélair and Mackey [7]):

$$\begin{aligned} y_1'(t) &= \hat{s}_0 y_2(t - T_1) - \gamma y_1(t) - Q, \\ y_2'(t) &= f(y_1(t)) - k y_2(t), \\ y_3'(t) &= 1 - \frac{Q e^{\gamma y_3(t)}}{\hat{s}_0 y_2(t - T_1 - y_3(t))}, \end{aligned}$$

where $f(y) = \frac{a}{1+Ky^r}$, $\hat{s}_0 = 0.0031$, $T_1 = 6$, $\gamma = 0.001$, $Q = 0.0275$, $k = 2.8$, $a = 6570$, $K = 0.0382$, $r = 6.96$, for t in $[0, 300]$. The history functions are

$$\begin{aligned} \phi_1(0) &= 3.325, \quad \text{for } \ t \leq 0 \\ \phi_2(t) &= \begin{cases} 10, & \text{for } \ -T_1 \leq t \leq 0 \\ 9.5, & \text{for } \ t < -T_1 \end{cases} \\ \phi_3(0) &= 120, \quad \text{for } \ t \leq 0. \end{aligned}$$

The exact solution of this problem is unknown.
The C^0 discontinuity of the history at $t = -T_1$ and the C^1 discontinuity at the starting point are propagated by both delays. The inherent and propagated C^{n+1} discontinuities at $t = nT_1$, $n \geq -1$, are mixed later with discontinuities caused by crossings of the state-dependent delay, with the first occurrence at $\lambda \approx 114.3085$ crossing $t = -T_1$.

The parameters are,

$$\mathbf{p} = [\hat{s}_0,\ T_1,\ \gamma,\ Q,\ k,\ a,\ K,\ r].$$

System 8. A scalar equation that exhibits chaotic behavior. It is an example of the well known Mackey-Glass delay differential equations which they proposed as a model for the production of white blood cells [6]. The problem has a constant delay and a constant history, and is defined by

$$y'(t) = \frac{2y(t-2)}{1+y(t-2)^{9.65}} - y(t),$$

for t in $[0, 100]$. The history function is

$$\phi(t) = 0.5, \quad \text{for } t \leq 0.$$

The exact solution of this problem is unknown.

The parameters are,

$$\mathbf{p} = [\tau,\ n,\ \mathrm{A}],$$

where

$$y'(t) = \frac{2y(t-\tau)}{1+y(t-\tau)^{n}} - y(t),$$

and

$$\phi(t) = \mathrm{A}, \quad \text{for } t \leq 0.$$

System 9. A triangle wave defined by the following NDDE,

$$y'(t) = -y'(t-\tau),$$

where $\tau = 0.1$, for t in $[0, 0.9]$. The history function is

$$\phi(t) = -t + \mathrm{c},$$

where $c = 8$, for $t \leq 0$.
The objective function is C^1 discontinuous for values of τ where a data point coincides with an integer multiple of τ.

The parameters are,

$$\mathbf{p} = [\tau,\ \mathrm{c}].$$

System 10. A Kermack-McKendrick model of an infectious disease with periodic outbreak ([1], [4], [11], [12]). The problem is defined by

$$\begin{aligned} y_1' &= -y_1(x)y_2(x-\tau) + y_2(x-\rho), \\ y_2' &= y_1(x)y_2(x-\tau) - y_2(x), \\ y_3' &= y_2(x) - y_2(x-\rho), \end{aligned}$$

where $\tau^\star = 1$ and $\rho^\star = 10$, for t in $[0, 55]$. The history function is

$$\begin{aligned} y_1 &= 5, \\ y_2 &= 0.1, \\ y_3 &= 1, \end{aligned}$$

for $t \leq 0$.
The exact solution of this problem is unknown.
Both delays could cause C^2 discontinuities in the objective function whenever a data point is at $t = \tau$ or $t = \rho$.

The parameters are,

$$\mathbf{p} = [\tau,\ \rho].$$

4.2 Driver Programs

A general driver programm is provided for each task (simulation, sensitivity analysis, parameter estimation) which can be used for a chosen system after slight modifications.

4.2.1 General Driver for Simulation

```
#include <iostream>
using namespace std;

#include <iomanip>

#include <math.h>

#include "ddem_system.h"
#include "ddem_simulator.h"
#include "ivp2dde_improved_CRK.h"
#include "ddem_sol_out.h"
#include "ddem_error_codes.h"

#include "SYSTEMS/systems.h"

/* only one of these is necessary */
#include "SYSTEMS/system_01.h"
#include "SYSTEMS/system_02.h"
#include "SYSTEMS/system_03.h"
#include "SYSTEMS/system_04.h"
#include "SYSTEMS/system_05.h"
#include "SYSTEMS/system_06.h"
```

```
#include "SYSTEMS/system_07.h"
#include "SYSTEMS/system_08.h"
#include "SYSTEMS/system_09.h"
#include "SYSTEMS/system_10.h"

#define round(x) ((int)floor((x) + 0.5))

int main(void)
{
    int experimentIdentifier = 8; /* used to select different
                              experiments conveniently = 1,2,3,4,5,6,7,8,... */

    ddemSystem* system;
    system = new ddemSystem();

    ddemSimulationAttributes* simA;


    double TOL = 1e-9;
    /* create the system and set all required functions and parameters*/
    switch(experimentIdentifier){
        case 1:
            getSystem_01(system);
            simA = new ddemSimulationAttributes(system->getNVariables(),
                                                system->getNParameters());
            getSimulationA_01(TOL, simA);
            break;
        case 2:
            getSystem_02(system);
            simA = new ddemSimulationAttributes(system->getNVariables(),
                                                system->getNParameters());
            getSimulationA_02(TOL, simA);
            break;
        case 3:
            getSystem_03(system);
            simA = new ddemSimulationAttributes(system->getNVariables(),
                                                system->getNParameters());
            getSimulationA_03(TOL, simA);
            break;
        case 4:
            getSystem_04(system);
            simA = new ddemSimulationAttributes(system->getNVariables(),
                                                system->getNParameters());
            getSimulationA_04(TOL, simA);
            break;
```

```
    case 5:
        getSystem_05(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_05(TOL, simA);
        break;
    case 6:
        getSystem_06(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_06(TOL, simA);
        break;
    case 7:
        getSystem_07(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_07(TOL, simA);
        break;
    case 8:
        getSystem_08(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_08(TOL, simA);
        break;
    case 9:
        getSystem_09(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_09(TOL, simA);
        break;
    case 10:
        getSystem_10(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_10(TOL, simA);
        break;
    default:
        cout << "Invalid experimentIdentifier " << endl;
        return -1;
}

int retCode;

retCode = system->checkIntegrity();
```

```
if(retCode != DDEM_TRUE){
    cout << "The DDE system is not well defined." << endl <<
              "Error Code = " << retCode << endl;
    return retCode;
}else
    cout << "The DDE system is well defined." << endl;



IVP2DDEImprovedCRK* improvedCRK;
improvedCRK = new IVP2DDEImprovedCRK(0/*interp_Flag*/);

ddemSimulator*  simulator;
simulator = new ddemSimulator(improvedCRK);

ddemSimulationSolution* simulationSolution;

simulationSolution = new ddemSimulationSolution();
retCode =
    simulator->simulate(system, simA->tEnd,
                                simA->parameters,
                                simA->relTolerance,
                                simA->absTolerance,
                                simA->vanishingConst,
                                simA->comm,
                                simulationSolution);

cout << endl << endl << "Return code = " << retCode << endl;
if(retCode != DDEM_SUCCESS)
    return retCode;

/*0000000000000000000000000000000000000000000000000000000000000000000000*/
int outYIndex = 0;/* The chosen component for output = 0,1,...,
                                            (nVariables - 1) */

int interpSelect = DDEM_INTERP;/* DDEM_INTERP = value required,
                                DDEM_DERIV_INTERP = derivative required*/

string fileNameBase;
string fileNameOutUniform;

fileNameBase = "OUT//experiment_";
fileNameBase.append(ddemInt2string(experimentIdentifier));
fileNameBase.append("_");
```

```
fileNameBase.append(ddemInt2string(round(-log10(TOL))));

fileNameOutUniform = fileNameBase;
fileNameOutUniform.append("_U.dat");

/* hight precision uniform output : suitable for error analysis, ... */
ddemOutSolUniform(simulationSolution,
                                        outYIndex,
                                        interpSelect,
                                        1000/*nOutPoints*/,
                                        fileNameOutUniform,
                                        16  /*precision*/,
                                        simulator);

/* low precision non-uniform : suitable for plots*/
string fileNameOutNonUniform;
string fileNameOutNonUniformDisc;

fileNameOutNonUniform = fileNameBase;
fileNameOutNonUniform.append("_NU.dat");

fileNameOutNonUniformDisc = fileNameBase;
fileNameOutNonUniformDisc.append("_DISC.dat");

//ddemOutSolNonUniform(simulationSolution,
//                                        outYIndex,
//                                        interpSelect,
//                                        1000/*nOutPoints*/,
//                                        fileNameOutNonUniform,
//                                        fileNameOutNonUniformDisc,
//                                        8  /*precision*/,
//                                        simulator);
ddemOutSolNonUniformAll(simulationSolution,
                                        interpSelect,
                                        1000/*nOutPoints*/,
                                        fileNameOutNonUniform,
                                        fileNameOutNonUniformDisc,
                                        8  /*precision*/,
                                        simulator);
/*-------------- display integration statistics -----------*/
cout << endl;
cout << setiosflags( ios::scientific | ios::showpoint ) << setprecision(4);
cout << "TOL  = " << simA->relTolerance << endl;

cout << endl;
```

```
cout << "Number of successful steps = "
     << simulationSolution->statistics->nSteps << endl;
cout << "Number of failed steps = "
     << simulationSolution->statistics->nFails << endl;
cout << "Number of function(right hand side) evaluations = "
     << simulationSolution->statistics->nFEvals << endl;
cout << "Number of vanishing steps = "
     << simulationSolution->statistics->nVanishingSteps << endl;
cout << "Number of iteraions caused by vanishing steps = "
     << simulationSolution->statistics->nIteraions4VanishingSteps << endl;

/* Display discontinuities along with their degree */
cout << setiosflags( ios::scientific | ios::showpoint ) << setprecision(15);
cout << endl << "Tracked Discontinuities:" <<endl;
for(int i=0; i < simulationSolution->nLambdas; i++)
    cout << "i = " << i << " lambda = " <<
    simulationSolution->meshPoints[simulationSolution->lambdasIndices[i]] <<
    " DiscDegree = " << simulationSolution->lambdasDiscDegrees[i] << endl;

/* Evaluete and display the solution values at the end point*/
double* y;
int m = simulationSolution->getNVariables();
y   = new double[m];
double t = simulationSolution->
            meshPoints[simulationSolution->nMeshPoints - 1]/* end point */;
simulator->evaluateSolution(t, y, simulationSolution, DDEM_INTERP,
                            DDEM_UNKNOWN);
cout << endl;
cout << "Computed values at the endpoint: ";
cout.unsetf(ios::scientific);
cout.setf(ios::fixed);
cout << setprecision(4);
cout << "t = " << t << endl;

cout.unsetf(ios::fixed);
cout << setiosflags( ios::scientific | ios::showpoint ) << setprecision(16);
for(int i=0; i < m; i++)
    cout << "y[" << i << "] = " << y[i] << endl;

/* Use exact values to report errors */
double* yExact;
yExact = new double[m];
if(experimentIdentifier == 1){
    double zeta2=4.0+2.0*log(2.0);
    yExact[0] = 4.0 - 2.0 * log(1.0+zeta2-t);
```

```
    }else if(experimentIdentifier == 2){
        yExact[0] = 0.3318616184531945;
        yExact[1] = 2.2222766635175959;
    }else if(experimentIdentifier == 3){
        yExact[0] = pow(exp(1.0)/(3.0-log(t)),exp(1.0));
    }else if (experimentIdentifier == 5){
        yExact[0] = pow(t,3.0 + simA->parameters[0]);
    }else if(experimentIdentifier == 4){
        yExact[0] = 7357.6215803296436000;
    }else if(experimentIdentifier == 6){
        yExact[0] = 5.2312724900129703;
        yExact[1] = 0.0549084622861634;
        yExact[2] = 3.9851129367680183;
        yExact[3] = 5.9156352730635131;
    }else if(experimentIdentifier == 7){
        yExact[0] = 3.4997442903907818;
        yExact[1] = 9.9999175130290432;
        yExact[2] = 1.1993285802286043e2;
    }else if(experimentIdentifier == 8){
        yExact[0] = 8.4025502940268892e-1;
    }
    cout << endl;
    cout << "Absolut and relative errors for different components:" << endl;
    cout.unsetf(ios::fixed);
    cout << setiosflags( ios::scientific | ios::showpoint ) << setprecision(16);
    for(int i=0; i < m; i++){
        double absError = fabs(y[i] - yExact[i]);
        double relError = absError / fabs(yExact[i]);
        cout << "absErr[" << i << "] = " << absError << "      " <<
                  "relErr[" << i << "] = " << relError << endl;
    }
    delete yExact;
    /*------*/
    delete y; y = NULL;
    /*--------------------*/
    delete simulationSolution;
    delete simulator;
    delete improvedCRK;
    delete simA;
    delete system;
    /*--------------------*/
  return 0;
}
```

We put the part of code that defines the system along with all its func-

tions and subroutines in a separate file. This is just because we found it an organized approach which makes it easier to alter a chosen system or extend the driver to simulate new systems. However, the user is free to put the system description code inside the main file or another preferred place. (See [14, Chapter 6] and [16] for the numerical results for “Systems 1” – “System 7”.)

The code for “System 1”” is attached below.

4.2.2 system_01.h

```
#ifndef __SYSTEM_01
#define __SYSTEM_01

#include "../ddem_system.h"
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
void f_01(int state, double t, int m, double* y,
                                          int nu, int omega, double* z,
                                            int l, double* p,
                                            double* f,
                      ddemUserProvidedJac* userProvidedJac,
                                            void* comm);

void delayArgumentsFunc_01(double t, int m, double* y,
                                   int l, double* p,
                                   int nu, int omega, double* alpha,
                                   ddemUserProvidedJac* userProvidedJac,
                                                                    void* comm);


void y0Func_01(int l, double* p,
                       int m, double* y0,
                       ddemUserProvidedJac* userProvidedJac,
                                             void* comm);

void getSystem_01(ddemSystem* system);

void getSimulationA_01(double TOL, ddemSimulationAttributes* simA);
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
#endif /*__SYSTEM_01*/
```

4.2.3 system_01.cc

```
#include "systems.h"
#include "system_01.h"
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
#ifndef __DDEM_INDEX_TRANSLATOR_DEF
#define __DDEM_INDEX_TRANSLATOR_DEF

#define Z(I,J) z[(J)*m + (I)]
/* Z(I,J) corresponds to y_I(alpha_J) if  0<= J < nu */
/* Z(I,J) corresponds to y'_I(alpha_J) if nu <= J < nu + omega */

#define TJAC(I) userProvidedJac->tJac[(I)]
#define YJAC(I,J) userProvidedJac->yJac[(I)*m + (J)]
#define ZJAC(I,J,K) userProvidedJac->zJac[(I)*(m*(nu+omega)) + (K)*m+(J)]
/* par F_I / par Z(J,K) */
#define PJAC(I,J) userProvidedJac->pJac[(I)*l + (J)]

#define TTJAC(I) userProvidedJac->ttJac[(I)]
#define TPJAC(I,J) userProvidedJac->tpJac[(I)*l + (J)]

#endif // __DDEM_INDEX_TRANSLATOR_DEF
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
void f_01(int state, double t, int m, double* y,
                                          int nu, int omega, double* z,
                                            int l, double* p,
                                            double* f,
                      ddemUserProvidedJac* userProvidedJac,
                                            void* comm)
{
    f[0] = Z(0,0);
    if(userProvidedJac->needed)
    {
        userProvidedJac->tIndependent = DDEM_TRUE;
        userProvidedJac->yIndependent = DDEM_TRUE;
        ZJAC(0,0,0) = 1.0;
        userProvidedJac->pIndependent = DDEM_TRUE;
    }
}
void delayArgumentsFunc_01(double t, int m, double* y,
                                    int l, double* p,
                                    int nu, int omega, double* alpha,
                                    ddemUserProvidedJac* userProvidedJac,
                                                                  void* comm)
```

```
{
  alpha[0] = y[0];
    if(userProvidedJac->needed)
    {
        userProvidedJac->tIndependent = DDEM_TRUE;
        YJAC(0,0) = 1.0;
        userProvidedJac->pIndependent = DDEM_TRUE;
    }
}
void y0Func_01(int l, double* p,
                       int m, double* y0,
                       ddemUserProvidedJac* userProvidedJac,
                                            void* comm)
{
    y0[0] = p[0];
    if(userProvidedJac->needed)
    {
        PJAC(0,0) = 1.0;
    }
}
/*&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&*/
void getSystem_01(ddemSystem* system)
{
    system->create(1, 1, 1, 0);

    system->setF(f_01);

    double h[]={0.5};
    system->setHistory(h);

    system->setT0(2.0);
    system->setT0DiscDegree(0);

    system->setY0(y0Func_01);

    system->setDelayArguments(1, 0, delayArgumentsFunc_01, DDEM_TRUE);
}
/*&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&*/
void getSimulationA_01(double TOL, ddemSimulationAttributes* simA)
{
    simA->parameters[0] = 1.0;

    simA->tEnd = 5.5;

    simA->relTolerance = TOL;
```

```
    simA->absTolerance[0] = TOL;
}
```

4.2.4 General Driver for Sensitivity Analysis

```
#include <iostream>
using namespace std;

#include <iomanip>

#include <math.h>

#include "ddem_system.h"
#include "ddem_fsens_system.h"
#include "ddem_simulator.h"
#include "ddem_sensitivity_analyzer.h"
#include "ivp2dde_improved_CRK.h"
#include "ddem_sol_out.h"
#include "ddem_sens_out.h"
#include "ddem_error_codes.h"

#include "SYSTEMS/systems.h"

/* only one of these is necessary */
#include "SYSTEMS/system_01.h"
#include "SYSTEMS/system_02.h"
#include "SYSTEMS/system_03.h"
#include "SYSTEMS/system_08.h"

enum {DDEM_EXP_SIM, DDEM_EXP_SENS, DDEM_EXP_PARA};

#define round(x) ((int)floor((x) + 0.5))

int main(void)
{
    int experimentIdentifier = 1; /* used to select different experiments
                            conveniently = 1,2,3,4*/


    ddemSystem* system;
    system = new ddemSystem();

    ddemSimulationAttributes* simA;
```

```
double TOL = 1e-9;
/* create the system and set all required functions and parameters*/
switch(experimentIdentifier){
    case 1:
        getSystem_01(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_01(TOL, simA);
        break;
    case 2:
        getSystem_02(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_02(TOL, simA);
        break;
    case 3:
        getSystem_03(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_03(TOL, simA);
        break;
    case 4:
        getSystem_08(system);
        simA = new ddemSimulationAttributes(system->getNVariables(),
                                            system->getNParameters());
        getSimulationA_08(TOL, simA);
        break;
    default:
        cout << "Invalid experimentIdentifier " << endl;
        return -1;
}

int retCode;

retCode = system->checkIntegrity();


if(retCode < 0){
    cout << "The DDE system is not well defined." << endl <<
              "Error Code = " << retCode << endl;
    return retCode;
}else
    cout << "The DDE system is well defined." << endl;
```

```
IVP2DDEImprovedCRK* improvedCRK;
improvedCRK = new IVP2DDEImprovedCRK(0/*interp_Flag*/);

ddemSimulator*  simulator;
simulator = new ddemSimulator(improvedCRK);


ddemSensitivitySolution* sensitivitySolution;
sensitivitySolution = new ddemSensitivitySolution();

ddemSensitivityAnalyzer* sensitivityAnalyzer;
sensitivityAnalyzer = new ddemSensitivityAnalyzer();

int* needSensitivity;
needSensitivity = new int[system->getNParameters()];
for(int l=0; l < system->getNParameters(); l++)
    needSensitivity[l] = DDEM_TRUE;

retCode =
    sensitivityAnalyzer->computeSensitivities(simulator,
                                system,
                                simA->tEnd,
                                simA->parameters,
                                simA->relTolerance,
                                simA->absTolerance,
                                simA->vanishingConst,
                                needSensitivity,
                                simA->comm,
                                sensitivitySolution);

cout << endl << endl << "Return code = " << retCode << endl;
if(retCode != DDEM_SUCCESS)
    return retCode;

ddemSensitivitySolutionFD* sensitivitySolutionFD;

sensitivitySolutionFD = new ddemSensitivitySolutionFD();

double deltaP = 1e-7;
retCode =
        sensitivityAnalyzer->computeSensitivitiesFD(simulator,
                                system,
                                simA->tEnd,
                                simA->parameters,
```

```
                                    deltaP,
                                    simA->relTolerance,
                                    simA->absTolerance,
                                    simA->vanishingConst,
                                    needSensitivity,
                                    1000,
                                    simA->comm,
                                    sensitivitySolutionFD);

delete needSensitivity;

cout << endl << endl << "Return code FD = " << retCode << endl;

/*0000000000000000000000000000000000000000000000000000000000000000000000000*/
int outYIndex = 0;/* The chosen component for output = 0,1,...,
                                         (nVariables - 1) */
int parameterIndex = 0;


int interpSelect = DDEM_INTERP;/* DDEM_INTERP = value required,
                                  DDEM_DERIV_INTERP = derivative required*/

string fileNameBase;
string fileNameOutUniform;
string fileNameOutUniformFD;

fileNameBase = "OUT//sens_experiment_";
fileNameBase.append(ddemInt2string(experimentIdentifier));
fileNameBase.append("_");
fileNameBase.append(ddemInt2string(round(-log10(TOL))));

fileNameOutUniform = fileNameBase;
fileNameOutUniform.append("_U.dat");
fileNameOutUniformFD = fileNameBase;
fileNameOutUniformFD.append("_");
fileNameOutUniformFD.append(ddemInt2string(round(-log10(deltaP))));
fileNameOutUniformFD.append("_U_FD.dat");

/* hight precision uniform output : suitable for error analysis, ... */
ddemOutSensUniform(sensitivitySolution,
                                        outYIndex,
                                        parameterIndex,
                                        interpSelect,
                                        1000/*nOutPoints*/,
                                        fileNameOutUniform,
```

```
                                        16  /*precision*/,
                                        simulator);

ddemOutSensFDUniform(sensitivitySolutionFD,
                                            outYIndex,
                                            parameterIndex,
                                            fileNameOutUniformFD,
                                            16 /*precision*/);

/* low precision non-uniform : suitable for plots*/
string fileNameOutNonUniform;
string fileNameOutNonUniformDisc;

fileNameOutNonUniform = fileNameBase;
fileNameOutNonUniform.append("_NU.dat");

fileNameOutNonUniformDisc = fileNameBase;
fileNameOutNonUniformDisc.append("_DISC.dat");

ddemOutSensNonUniformAll(sensitivitySolution,
                                        interpSelect,
                                        1000/*nOutPoints*/,
                                        fileNameOutNonUniform,
                                        fileNameOutNonUniformDisc,
                                        8  /*precision*/,
                                        simulator);
/*-------------- display integration statistics -----------*/
cout << endl;
cout << setiosflags( ios::scientific | ios::showpoint ) << setprecision(4);
cout << "TOL  = " << simA->relTolerance << endl;

cout << endl;
cout << "Number of successful steps = "
     << sensitivitySolution->statistics->nSteps << endl;
cout << "Number of failed steps = "
     << sensitivitySolution->statistics->nFails << endl;
cout << "Number of function(right hand side) evaluations = "
     << sensitivitySolution->statistics->nFEvals << endl;
cout << "Number of vanishing steps = "
     << sensitivitySolution->statistics->nVanishingSteps << endl;
cout << "Number of iteraions caused by vanishing steps = "
     << sensitivitySolution->statistics->nIteraions4VanishingSteps << endl;

/* Display discontinuities along with their degree */
cout << setiosflags( ios::scientific | ios::showpoint ) << setprecision(15);
```

```
    cout << endl << "Tracked Discontinuities:" <<endl;
    for(int i=0; i < sensitivitySolution->nLambdas; i++)
        cout << "i = " << i << " lambda = " <<
        sensitivitySolution->meshPoints[sensitivitySolution->lambdasIndices[i]] <<
        " DiscDegree = " << sensitivitySolution->lambdasDiscDegrees[i] << endl;
    /*-------------------*/
    delete sensitivitySolution;
    delete sensitivitySolutionFD;
    delete simulator;
    delete sensitivityAnalyzer;
    delete improvedCRK;
    delete simA;
    delete system;
    /*-------------------*/
  return 0;
}
```

See [14, Chapter 6] and [15] for the numerical results and discussions regarding the following test cases.

Test Case 1 Sensitivity of the solution with respect to a discontinuity at the initial point for "System 1".

Test Case 2 Sensitivity of the solution with respect to all parameters and histories for "System 2".

Test Case 3 Sensitivity of the solution with respect to the starting time for "System 3".

Test Case 4 Sensitivity of the solution with respect to the delay, exponent and history for "System 8".

4.2.5 General Driver for Parameter Estimation

```
#include <iostream>
using namespace std;
#include <fstream>
#include <iomanip>
#include <string>

#include <math.h>
#include <stdlib.h>
#include <time.h>

#include "ddem_system.h"
```

```
#include "ddem_simulator.h"
#include "ivp2dde_improved_CRK.h"
//#include "ddem_sensitivity_analyzer.h"
//#include "ddem_sol_out.h"

#include "ddem_parameter_estimator.h"

#include "SYSTEMS/systems.h"

/* only one of these is necessary */
#include "SYSTEMS/system_01.h"
#include "SYSTEMS/system_02.h"
#include "SYSTEMS/system_09.h"
#include "SYSTEMS/system_10.h"

#define my_rand(min, max) (int) (min + ((max) - (min)) * rand()/((double)RAND_MAX + 1))

void get_data_and_constraints_01(int nParameters, ddemData* data,
                                 ddemConstraints* constraints);
void get_data_and_constraints_02(int nParameters, ddemData* data,
                                 ddemConstraints* constraints);
void get_data_and_constraints_09(int nParameters, ddemData* data,
                                 ddemConstraints* constraints);
void get_data_and_constraints_10(int nParameters, ddemData* data,
                                 ddemConstraints* constraints);


//#include <stdio.h>
int main(void)
{

  int experimentIdentifier = 4; /* used to select different experiments     */
                                /* conveniently = 1,2,3,4                   */
                                /*      corresponding systems are: 9,1,10,2 */

  int nRuns = 1;

  int solvers[] ={DDEM_PE_VERY_SIMPLE, DDEM_PE_SIMPLE, DDEM_PE_NEW, DDEM_PE_NEW_CE};
  /*                  0                   1             2            3          */
  int selectedSolver = 3;


  ddemSystem* system;
  system = new ddemSystem();
```

```
ddemSimulationAttributes* simA;

ddemData* data;
data = new ddemData();

ddemConstraints* constraints;
constraints = new ddemConstraints();

double TOL = 1e-6;
/* create the system and set all required functions and parameters*/
switch(experimentIdentifier){
  case 1:
    getSystem_09(system);
    simA = new ddemSimulationAttributes(system->getNVariables(),
                                        system->getNParameters());
    getSimulationA_09(TOL, simA);
    get_data_and_constraints_09(system->getNParameters(), data, constraints);
    break;
  case 2:
    getSystem_01(system);
    simA = new ddemSimulationAttributes(system->getNVariables(),
                                        system->getNParameters());
    getSimulationA_01(TOL, simA);
    get_data_and_constraints_01(system->getNParameters(), data, constraints);
    break;
  case 3:
    getSystem_10(system);
    simA = new ddemSimulationAttributes(system->getNVariables(),
                                        system->getNParameters());
    getSimulationA_10(TOL, simA);
    get_data_and_constraints_10(system->getNParameters(), data, constraints);
    break;
  case 4:
    getSystem_02(system);
    simA = new ddemSimulationAttributes(system->getNVariables(),
                                        system->getNParameters());
    getSimulationA_02(TOL, simA);
    get_data_and_constraints_02(system->getNParameters(), data, constraints);
    break;
  default:
    cout << "Invalid experimentIdentifier " << endl;
    return -1;
}

int retCode;
```

```
retCode = system->checkIntegrity();


if(retCode < 0){
  cout << "The DDE system is not well defined." << endl <<
          "Error Code = " << retCode << endl;
  return retCode;
}else
  cout << "The DDE system is well defined." << endl;
/*-----------------------------------------------*/
/*-----------------------------------------------*/
IVP2DDEImprovedCRK* improvedCRK;
improvedCRK = new IVP2DDEImprovedCRK(0/*interp_Flag*/);

ddemSimulator*  simulator;
simulator = new ddemSimulator(improvedCRK);

/*-----------------------------------------------*/
srand( (unsigned)time( NULL ) );

double* perturbedOriginalParameters;
perturbedOriginalParameters = new double[system->getNParameters()];
double* optimumParameters;
optimumParameters = new double[system->getNParameters()];
/*-----------------------------------------------*/
double timeTotal[4];
int nFEvalsTotal[4];
int nReferencedParameterValuesTotal[4];
double finalObjectiveValueTotal[4];
double parameterDiffNormTotal[4];
for(int solverID = 0; solverID < 4; solverID++){
  timeTotal[solverID] = 0.0;
  nFEvalsTotal[solverID] = 0;
  nReferencedParameterValuesTotal[solverID] = 0;

  finalObjectiveValueTotal[solverID] = 0.0;
  parameterDiffNormTotal[solverID] = 0.0;
}


for(int runCounter = 1; runCounter <= nRuns; runCounter++){
  cout << "********************************** runCounter = " <<
    runCounter  << "**********************************"<< endl;
  for(int i=0; i < system->getNParameters(); i++){
```

```
  if(constraints->isFree(i) == DDEM_TRUE)
      perturbedOriginalParameters[i] = simA->parameters[i] *
                                          (1.0 + my_rand(-10,10)/100.0);
    else
      perturbedOriginalParameters[i] = simA->parameters[i];
    optimumParameters[i] = perturbedOriginalParameters[i];
}
for(int solverID = 0; solverID < 4; solverID++){ // run all the solvers
//for(int solverID = selectedSolver; solverID < selectedSolver + 1; solverID++){
  cout << "--------------------------------" << "SOLVER ID = " << solverID <<
    "--------------------------------------"<< endl;
  ddemParameterEstimator* parameterEstimator;
  parameterEstimator = new ddemParameterEstimator();
  ddemParameterEstimationStatistics* statistics;
  statistics  = new  ddemParameterEstimationStatistics();

  clock_t start, end;
  double cpu_time_used;

  start = clock();/*TIME TIME TIME TIME*/
  int retCodeTemp =
  parameterEstimator->EstimateParameters(simulator, system, data,
                                         constraints,
                                         optimumParameters,
                                         simA->relTolerance,
                                         simA->absTolerance,
                                         simA->vanishingConst,
                                         statistics,
                                         solvers[solverID],
                                         simA->comm);
  end = clock();/*TIME TIME TIME TIME*/
  cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
  timeTotal[solverID] += cpu_time_used;

  cout << "return code = " << retCodeTemp << "\n";

  cout << "nFEvals = " << statistics->nFEvals << endl;
  nFEvalsTotal[solverID] += statistics->nFEvals;
  cout << "nReferencedParameterValues = " <<
           statistics->nReferencedParameterValues << endl;
  nReferencedParameterValuesTotal[solverID] +=
                                    statistics->nReferencedParameterValues;

  cout << "Final Objective Value = " << statistics->finalObjectiveValue << endl;
  finalObjectiveValueTotal[solverID] += statistics->finalObjectiveValue;
```

```
    double parameterDiffNorm = 0.0;
    for(int i=0; i < system->getNParameters(); i++)
      parameterDiffNorm = ddem_max(parameterDiffNorm,
                                   fabs(perturbedOriginalParameters[i] -
                                          optimumParameters[i]));

    cout << "Maximum Parameter Difference = " << parameterDiffNorm << endl;
    parameterDiffNormTotal[solverID] += parameterDiffNorm;

    cout << endl;
    for(int i=0; i < system->getNParameters(); i++){
      cout << "originalParameters[" << i << "] = ";
      cout.setf(ios::fixed); cout << setprecision(16);
      cout << simA->parameters[i] << "\n";
    }
    cout << endl;
    for(int i=0; i < system->getNParameters(); i++){
      cout << "perturbedOriginalParameters[" << i << "] = ";
      cout.setf(ios::fixed); cout << setprecision(16);
      cout << perturbedOriginalParameters[i] << "\n";
    }
    cout << endl;
    for(int i=0; i < system->getNParameters(); i++){
      cout << "optimumParameters[" << i << "] = ";
      cout.setf(ios::fixed); cout << setprecision(16);
      cout << optimumParameters[i] << "\n";
    }
    delete parameterEstimator;
    delete statistics;
  }/* solverID */
}/* runCounter */
for(int solverID = 0; solverID < 4; solverID++){ // run all the solvers
//for(int solverID = selectedSolver; solverID < selectedSolver + 1; solverID++){
  cout << "----------------------------------------" << "SOLVER ID = " <<
         solverID <<
    "---------------------------------------"<< endl;

  cout << "Average Values in " << nRuns << " runs" << endl << endl;
  cout << "nFEvals = " << (int)(nFEvalsTotal[solverID] / (double)nRuns) << endl;
  cout << "nReferencedParameterValues = " <<
              nReferencedParameterValuesTotal[solverID] / (double)nRuns << endl;

  cout << "Final Objective Value = " << finalObjectiveValueTotal[solverID] /
```

```
                                                    (double)nRuns << endl;
    cout << "Time = " << timeTotal[solverID] / (double)nRuns << endl;
  }

  delete optimumParameters;
  delete perturbedOriginalParameters;
  delete simulator;
  delete improvedCRK;

  delete constraints;
  delete data;
  delete system;
  delete simA;

  return 0;
}

/*************************************************************/
void get_data_and_constraints_01(int nParameters, ddemData* data,
                        ddemConstraints* constraints)
{
  string dataFileName;

  dataFileName = "IN//example_01.d"; // change

  data->Load(dataFileName);

  int nLinearConstraints;
  int nNonlinearConstraints;
  double infinityBound;

  nLinearConstraints = 0;// change
  nNonlinearConstraints = 0; // change
  infinityBound = 1e+20; // change or leave

  constraints->create(nParameters, nLinearConstraints, nNonlinearConstraints,
                                                       infinityBound);


  constraints->setSimpleLowerBound(0, .01);
  constraints->setSimpleUpperBound(0, 1.9);
}
/*************************************************************/
void get_data_and_constraints_02(int nParameters, ddemData* data,/
                        ddemConstraints* constraints)
```

```
{
  string dataFileName;

  dataFileName = "IN//example_02.d"; // change

  data->Load(dataFileName);

  int nLinearConstraints;
  int nNonlinearConstraints;
  double infinityBound;

  nLinearConstraints = 0;// change
  nNonlinearConstraints = 0; // change
  infinityBound = 1e+20; // change or leave

  constraints->create(nParameters, nLinearConstraints, nNonlinearConstraints,
                                                       infinityBound);
  constraints->setSimpleEquality(3, 33.0 / 100.0);
  constraints->setSimpleEquality(4, - 1.0 / 10.0);
  constraints->setSimpleEquality(5, 111.0 / 50.0);
  constraints->setSimpleEquality(6, 1.0 / 10.0);

  constraints->setSimpleLowerBound(0, .05);
}
/*************************************************************/
void get_data_and_constraints_09(int nParameters, ddemData* data,
                        ddemConstraints* constraints)
{
  string dataFileName;

  dataFileName = "IN//example_09.d";/* CHANGE */

  data->Load(dataFileName);

  int nLinearConstraints;
  int nNonlinearConstraints;
  double infinityBound;

  nLinearConstraints = 0;/* FILL */
  nNonlinearConstraints = 0; /* FILL */
  infinityBound = 1e+20; /* CHANGE */

  constraints->create(nParameters, nLinearConstraints, nNonlinearConstraints,
                                                       infinityBound);
```

```
  constraints->setSimpleLowerBound(0, .05);
  constraints->setSimpleEquality(1, 8.0);
}
/*************************************************************/
void get_data_and_constraints_10(int nParameters, ddemData* data,
                         ddemConstraints* constraints)
{
  string dataFileName;

  dataFileName = "IN//example_10.d"; // change

  data->Load(dataFileName);

  int nLinearConstraints;
  int nNonlinearConstraints;
  double infinityBound;

  nLinearConstraints = 0;// change
  nNonlinearConstraints = 0; // change
  infinityBound = 1e+20; // change or leave

  constraints->create(nParameters, nLinearConstraints, nNonlinearConstraints,
                                                       infinityBound);

  constraints->setSimpleLowerBound(0, .05);
  constraints->setSimpleLowerBound(1, 5.0);
}
/*************************************************************/
```

See [14, Chapter 6] for the numerical results and discussions regarding the following test cases.

Test Case 1 Estimating the delay for "System 9".

Test Case 2 Estimating $y(2)$ for "System 1".

Test Case 3 Estimating the delays for "System 10".

Test Case 4 Estimating structure-related parameters τ, ρ, α for "System 2".

Chapter 5

Tracing and Error Codes

By default a module in DDEM provides a return code indicating the situation after the return which is either a success code in case of performing the required task or an error code which then is interpreted by the user to find possible issues. However, DDEM gives the user an option for detailed monitoring of the progress of the task by providing outputs during the run. The header file `ddem_trace.h` can be modified by the user to turn the output on or off during the execution by changing the corresponding definitions.

5.0.6 ddem_trace.h

```
/* Used for trace with output 0 = notrace  1 = trace */
#ifndef __DDEM_TRACE
#define __DDEM_TRACE
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/

#define DDEM_TRACE_SYSTEM 0
#define DDEM_TRACE_MIN_ROOT 0
#define DDEM_TRACE_SIMULATOR 0
#define DDEM_TRACE_SENSITIVITY_ANALYZER 0
#define DDEM_TRACE_DATA 0
#define DDEM_TRACE_PARAMETER_ESTIMATOR_DETAILED 0
#define DDEM_TRACE_PARAMETER_ESTIMATOR 0

/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
#endif /*__DDEM_TRACE*/
```

5.0.7 ddem_error_codes.h

```
#ifndef __DDEM_ERROR_CODES
```

```
#define __DDEM_ERROR_CODES
/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
#define DDEM_SUCCESS 0
#define DDEM_FAILURE -1

#define DDEM_ERR_BASE                                   (-50)
#define DDEM_ERR_UNEXPECTED                             (DDEM_ERR_BASE - 1)

/*SIMULATOR*/
#define DDEM_ERR_SIM_BASE                               (-100)
#define DDEM_ERR_SIM_MAX_MESH_POINTS_REACHED            (DDEM_ERR_SIM_BASE - 1)
#define DDEM_ERR_SIM_MAX_DISC_POINTS_REACHED            (DDEM_ERR_SIM_BASE - 2)
#define DDEM_ERR_SIM_ADV_OR_VANISHING_DELAY             (DDEM_ERR_SIM_BASE - 3)

#define DDEM_ERR_SIM_T_LESS_THAN_T0                     (DDEM_ERR_SIM_BASE - 4)
#define DDEM_ERR_SIM_T_GREATER_THAN_TLAST               (DDEM_ERR_SIM_BASE - 5)

#define DDEM_ERR_SIM_LAMBDA_INDEX_OUT_OF_RANGE          (DDEM_ERR_SIM_BASE - 6)
#define DDEM_ERR_SIM_MESH_POINT_INDEX_OUT_OF_RANGE      (DDEM_ERR_SIM_BASE - 7)

#define DDEM_ERR_SIM_REQUIRED_EXTENSION_IS_IMPOSSIBLE   (DDEM_ERR_SIM_BASE - 7)
#define DDEM_ERR_SIM_CALL_DENIED                        (DDEM_ERR_SIM_BASE - 8)

#define DDEM_ERR_SIM_VANISHING_ITER_CANNOT_CONVERGE     (DDEM_ERR_SIM_BASE - 9)

/*SYSTEM*/
#define DDEM_ERR_SYS_BASE                               (-200)
#define DDEM_ERR_SYS_INVALID_HISTORY_SEGMENT_INDEX      (DDEM_ERR_SYS_BASE - 1)

/*MIN_ROOT*/
#define DDEM_ERR_MINROOT_BASE                           (-300)
#define DDEM_ERR_MINROOT_NO_ROOTS_WERE_FOUND            (DDEM_ERR_MINROOT_BASE - 4)

/*SENSITIVITY_ANALYZER*/
#define DDEM_ERR_SENS_BASE                              (-400)
#define DDEM_ERR_SENS_VANISHING_DENOMINATOR             (DDEM_ERR_SENS_BASE - 1)
#define DDEM_ERR_SENS_DATA_OUT_OF_RANGE                 (DDEM_ERR_SENS_BASE - 2)

/*PARAMETER_ESTIMATOR*/
#define DDEM_ERR_PARAMEST_BASE                          (-500)
#define DDEM_ERR_PARAMEST_MAX_ITER_REACHED              (DDEM_ERR_PARAMEST_BASE - 1)


/*NAG*/
#define DDEM_ERR_NAG_BASE                               (-1000)
```

```
/*IVP2DDE*/
#define DDEM_ERR_IVP2DDE_BASE                           (-2000)

/*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*/
#endif /*__DDEM_ERROR_CODES*/
```

Bibliography

[1] W.H. Enright. Software for Delay Differential Equations: Accurate Approximate Solutions are not Enough. Manuscript for Voltera 2004.

[2] W.H. Enright and L. Yan. The Quality/Cost Trade-off for a Class of ODE Solvers. *Numerical Algorithms*, DOI 10.1007/s11075-009-9288-x, 2009.

[3] L. Genik and P. van den Driessche. An Epidemic Model with Recruitment-Death Demographics and Discrete Delays. In S. Ruan, G.S.K. Wolkowicz, and J. Wu, editors, *Differential equations with applications to biology*, number 21 in Fields Institute Communications, pages 237–249. co-publication of the AMS and Fields Institute, Providence, RI, 1999.

[4] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff problems*. Springer Series in Computational Mathematics. Springer-Verlag, New York, second edition, 1993.

[5] Y. Kuang. On neutral delay logistic Gause-type predator-prey systems. *Dynamics and Stability of Systems*, 6:173–189, 1991.

[6] M.C. Mackey and L. Glass. Oscillation and Chaos in Physiological Control Systems. *Science*, 197(4300):287–289, 1977.

[7] J.M. Mahaffy, J. Bélair, and M.C. Mackey. Hematopoietic Model with Moving Boundary Condition and State Dependent Delay: Applications in Erythropoiesi. *J. Theor. Biol.*, 190:135–146, 1998.

[8] K.W. Neves. Automatic Integration of Functional Differential Equations: An Approach. *ACM Trans. Math. Soft.*, 1(4):357–368, 1975.

[9] C.A.H. Paul. Developing a delay differential equation solver. *Appl. Numer. Math.*, 9:403–414, 1992.

[10] C.A.H. Paul. *Runge-Kutta Methods for Functional Differential Equations*. PhD thesis, Manchester Univ., England, 1992.

[11] L.F. Shampine and S. Thompson. Solving DDEs in MATLAB. Manuscript, available at http://www.cs.runet.edu/∼thompson/webddes/ddepap.html, 2000.

[12] L.F. Shampine and S. Thompson. Solving Delay Differential Equations with dde23. Manuscript, available at http://www.runet.edu/∼thompson/webddes/index.html, 2000.

[13] L. Tavernini. The Approximate Solution of Volterra Differential Systems with State-Dependent Time Lags. *SIAM J. Numer. Anal.*, 15(5):1039–1052, 1978.

[14] H. Zivaripiran. *Efficient Simulation, Accurate Sensitivity Analysis and Reliable Parameter Estimation for Delay Differential Equations*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, 2009.

[15] H. Zivaripiran and W.H. Enright. Accurate First-Order Sensitivity Analysis for Delay Differential Equations: Part I: The Forward Approach. preprint, Department of Computer Science, University of Toronto, 2009.

[16] H. Zivaripiran and W.H. Enright. An Efficient Unified Approach for the Numerical Solution of Delay Differential Equations. *Numerical Algorithms*, 53(2), DOI 10.1007/s11075-009-9331-y:397–417, 2010.