ODEMx Documentation

Version 1.0 Humboldt-Universität Berlin
16. January 2004

Author: Ralf Gerstenberger

Table of Contents

Introduction

Welcome to ODEMx. ODEMx is a library for process simulation in C++. ODEMx is a C++ library for process simulation. Process simulation is a branch of computer simulation. As such, it uses processes to model real world or fictional systems. A process in this context is a continuous or discrete sequence of actions, which are somehow closely related to each other, for instance they are all 'done' by one agent. The sequence can be divided into branches, and broken off by idle periods or synchronisation.

Copyright and License

Installation

There is no installation procedure for ODEMx, at least in this version. To use ODEMx in your projects choose the respective folder that fits your needs best (Gcc on Unix systems, Msvc or Msvc7.1 on windows), build the library and finally set-up your project to use the library.

Note:

- If you are using GCC we recommend to use a version higher than 3.x.x . ODEMx has been successfully build with GCC version 2.95.2 and tested on Sun Sparc architecture. However, ODEMx has a problem with GCC prior to GCC version 3 on Linux (SuSE, Debian).
- Always take care to activate RTTI in your projects, because ODEMx depends on this feature. (MSVC and MSCVC7.1 disable RTTI by default)

First Steps

At first you will have to build ODEMx - see Installation for details. Afterwards try the examples included. That should give you a good start. Apart from that you can use them to check whether ODEMx is working properly on your computer.
ODEMx includes an online documentation as well. You will need Doxygen (http://www.doxygen.org) to generate the documentation from the source files. To do so, change to `odemx/Doxygen` directory and run `doxygen`.

Contributions

ODEMx is based on ODEM (http://odem.sf.net). That's why all contributors of ODEM could be listed here, too. Instead, only those who put their hands on this code directly will be mentioned. Anyway, we encourage you to look at ODEM and its contributors list as well.

Ralf Gerstenberger (gerstenb@users.sourceforge.net)

ODEMx - Overview

In this chapter, we will give an overview of ODEMx.

The Structure

ODEMx is divided into the six modules Base, Utilities, Random, Synchronisation, Statistic and Coroutine. All modules depend on Utilities. The Base module additionally depends on Coroutine and Random, while the Synchronisation module depends on Base. Utilities contains utility classes and interfaces of ODEMx. Coroutine provides a coroutine implementation. Base contains basic modelling concepts. The Random module is a collection of random number generators. Synchronisation and Statistic provide classes for synchronisation and statistical analysis.

Base

Base contains classes for the basic modelling concepts of process simulation and three important output classes. Not all classes in this module are for public use. Those of immediate relevance to a user of ODEMx are: `Simulation`, `Process`, `Continuous`, `HtmlTrace`, `HtmlReport` and `ContuTrace` which are declared and implemented in the files `Simulation.h`, `Simulation.cpp`, `Process.h`, `Process.cpp`, `Continuous.h`, `Continuous.cpp`, `HtmlTrace.h`, `HtmlTrace.cpp`, and finally `HtmlReport.h` along with `HtmlReport.cpp`.

Simulation

For every simulation with ODEMx, a class derived form the abstract base class Simulation is necessary. The Simulation class provides a context for all the other model components like processes, synchronisation classes and random number generators. In the user defined classes the `initSimulation()` method has to be implemented to create and initialise model components.
For convenience, ODEMx also provides the class `DefaultSimulation` with an empty implementation of `initSimulation()`. If the user does not want to define a specific simulation class, this default implementation can be used. A pointer to an object of this class is returned by the function `getDefaultSimulation()`. This function creates the object if required. ODEMx provides at most one object of `DefaultSimulation` in a program.
If a user defines their own simulation classes, several simulations in one program are possible. This includes simulations inside simulations and parallel simulations.

Documentation:
`odemx::Simulation`
`odemx::DefaultSimulation`
`odemx::getDefaultSimulation`

Process

`Process` is the base class for all user-defined classes of discrete processes. For every process in a simulation, the user has to specify a process class. This class has to provide the behaviour (the sequence of actions, synchronisation and idle periods) of its objects as an implementation of the function `main()`. While a process class is not necessarily associated to a simulation class, every process is linked to one simulation. (The constructor of the base class `Process` requires a pointer to a simulation.)

Documentation:
`odemx::Process`
`odemx::Process::main`

Continuous

As `Process` is the base class for discrete processes, `Continuous` is the base class for continuous processes. The class `Continuous` is derived from `Process`. It provides additional functions for the definition of time continuous behaviour. Time continuous behaviour can only be approximated by ODEMx. This is done through a step by step computation of state changes, which of course introduces an inherent error. ODEMx is observing this error to keep it in defined boundaries.

A user defined class of continuous processes has to provide an implementation of the function `derivatives()` in addition to its implementation of `main()`. In this function, the continuous state changes are coded by setting the change rates for each state variable. A continuous phase is than started in `main()` by a call of `integrate()`.

Documentation:
`odemx::Continuous`
`odemx::Continuous::integrate`
`odemx::Continuous::derivatives`

HtmlTrace

During development, it is often required to look at what actually happened during a simulation. `HtmlTrace` logs simulation events for this and other reasons. The backbone of `HtmlTrace` actually is a sophisticated trace system that supports multiple trace clients (like `HtmlTrace`) with different outputs (for example `Xml` instead of `Html`) fed by arbitrary trace producers. A user can define its own trace producers and add its specific events to the trace. The amount of events during a simulation is usually enormous. To reduce the number of logged events `HtmlTrace` provides a simple filter.

Documentation:
`odemx::HtmlTrace`
`odemx::HtmlTrace::setFilter`
`odemx::Trace`

HtmlReport

Like `HtmlTrace`, `HtmlReport` is used to gather information about a simulation. Instead of logging all simulation events, `HtmlReport` is used to report summaries of a simulation. Such summaries are often statistical analysis of samples from model components. A summary could for instance include the number of uses of a random number generator, the maximum time of processes spent in a queue or the average number of free token in a resource. ODEMx also supports multiple reports about different sets of model components.

Documentation:
`odemx::HtmlReport`
`odemx::Report`

ContuTrace

`ContuTrace` finally is a class for observing `Continuous` objects. It logs the state changes of continuous processes. The output is written in a text file. An object of the type `ContuTrace` is always linked to exactly one object of the type `Continuous`.

Documentation:
`odemx::ContuTrace`

Utilities

This module contains utility classes and interfaces for concepts of ODEMx. The Trace system as well as the base classes for `HtmlReport` can be found in this module. However, many classes in Utilities are not used directly. A user might be interested in the functions in `ErrorHandling.h` and in `odemx::Version`. For background information, it will be necessary to take a look at this module as well.

Random

Random provides a set of random number generators (RNG). There are RNG of different distributions for discrete and for continuous random numbers. This module is further divided into a module for Integer distributions and a module for continuous distributions. The usage of the RNG from these modules is quite similar. All provide a function `sample()` which returns the next random number. The parameters for the generators are set in the constructors.

Synchronisation

An important part of process simulation is to synchronise different processes with certain situations during a simulation. Synchronisation is a collection of classes, which provide model components for synchronisation

tasks. There are two resource-type synchronisation classes: `Bin` and `Res`, two queue-like synchronisation components `Waitq` and `Condq` and finally one synchronisation-point-type class `Wait`.

Bin and Res

`Bin` and `Res` are both used to model resource-like synchronisation. A process is blocked if the resource is exhausted otherwise it continuous its actions. `Bin` and `Res` use tokens to simulate real resources. A process takes and gives, or acquires and returns tokens.

Documentation:
`odemx::Bin`
`odemx::Res`

Waitq

`Waitq` provides a master-slave-synchronisation (MSS). A MSS is a synchronisation between two processes. One of them (the master) takes control after a successful synchronisation. The other (the slave) is delivered to the master.

Documentation:
`odemx::Waitq`

Condq

`Condq` is used to wait for an arbitrary condition. A process provides a function, which implements the condition check. The condition is checked every time a process signals a `Condq` object. The user is responsible to signal a `Condq` when the situation has changed.

Documentation:
`odemx::Condq`

Wait

The class `Wait` synchronises a process with a set of partner processes. The process that creates the `Wait` class is blocked until one or all of its partner processes are terminated.

Documentation:
`odemx::Wait`

Statistics

An important part of every simulation is to analyse the results. ODEMx supports this with six classes in the module Statistic: `Count`, `Sum`, `Tally`, `Accum`, `Histo` and finally `Regress`. These classes are used to collect and analyse samples. All of them have a usage counter and allow resetting their statistics.

Coroutine

Unlike the other modules, this module is not meant to be used by the user directly. It contains an implementation of coroutines, which fits the needs of ODEMx. The implementation is independent of Base and the other modules with the exception of some classes in Utilities.

From ODEM to ODEMx

In this chapter, we describe differences between ODEMx and its predecessor ODEM and features they have in common.

Overview

The following table lists all classes from ODEM and ODEMx and their status in ODEMx. This Table is not supposed to be documentation for ODEMx.

Class	Status	Description	Comment
Accum	Changed	Statistics	Statistics
Bin	Changed	Resource like synchronisation	Synchronisation
BinObserver	new	Observer	Observation
Buff_head	lost	Buffered communication	not transferred to ODEMx
Buff_tab	lost	Buffered communication	not transferred to ODEMx
Buff_tail	lost	Buffered communication	not transferred to ODEMx
Condq	Changed	Condition queue	Synchronisation
CondqObserver	new	Observer	Observation
Continuous	Changed	Time-continuous process	Continuous
ContinuousObserver	new	Observer	Observation
ContuTrace	new	Trace for Continuous	Continuous
Coroutine	new	Portable coroutine implementation	Coroutine
CoroutineContext	new	Portable coroutine implementation	Coroutine
CoroutineContextObserver	new	Observer	Observation
CoroutineObserver	new	Observer	Observation
Count	changed	Statistic	Statistics
DebugTrace	lost	Text-log	Replaced by HtmlTrace
DefaultContext	new	Portable coroutine implementation	Coroutine
DefaultOrder	new	Process sorting scheme	Process queue
DefaultSimulation	new	Default implementation of Simulation	Encapsulation
DefaultTimeIO	lost	Time to string to time	not transferred
DefLabeledObject	new	Object labels	Object labels
Discrete	lost	Time-discrete process	Replaced by Process
Dist	changed	Random number generator	Random
DistContext	new	Random number generator	Random
Draw	changed	Random number generator	Random
DynTableDefinition	new	Report	Report
Elem	lost	ODEM-internal	not transferred
Empirical	lost	Random number generator	not transferred
Entity	lost	ODEM-internal	not transferred
Entry	lost	ODEM-internal	not transferred
Erlang	changed	Random number generator	Random
Event	lost	ODEM-internal	not transferred
ExecutionList	new	Process scheduling	Process
ExecutionListObserver	new	Observer	Observation
File_list	lost	ODEM-internal	not transferred
FormatedTimeInput	lost	String to time	not transferred
FormatedTimeOutput	lost	Time to string	not transferred
Graph	lost	ODEM trace for Continuous	not transferred
Head	lost	ODEM-internal	not transferred
Histo	changed	Statistic	Statistics
HtmlReport	new	Report	Report
HtmlTrace	new	Trace	Trace
Iconst	changed	Random number generator	Random
Idist	changed	Random number generator	Random

LabeledObject	new	Object labels	Object labels
LabelScope	new	Unique object labels	Object labels
Link	lost	ODEM-internal	not transferred
MarkType	new	Trace	Trace
Memo	lost	ODEM object linking	Replaced by Observation
Msg	lost	Buffered communication	not transferred
Negexp	changed	Random number generator	Random
Normal	changed	Random number generator	Random
NoQueue	lost	ODEM-internal	not transferred
NoTally	lost	ODEM-internal	not transferred
Observable	new	Observation of individual objects	Observation
Object_names	lost	Object labels	Replaced by LabeledObject
Odem	lost	ODEM-internal	not transferred
Poisson	changed	Random number generator	Random
Port	lost	Port synchronisation	not transferred
Port_head	lost	Port synchronisation	not transferred
Port_tail	lost	Port synchronisation	not transferred
PriorityOrder	new	Process sorting scheme	Process queue
Process	changed	Process	Process
Process_clock	lost	Time event	not transferred
ProcessObserver	new	Observer	Observation
ProcessOrder	new	Process sorting scheme	Process queue
ProcessQueue	new	Process list	Process queue
Queue	changed	Process synchronisation queue	Synchronisation
Randint	changed	Random number generator	Random
Rconst	changed	Random number generator	Random
Rdist	changed	Random number generator	Random
Regress	changed	Statistics	Statistics
Report	new	Report	Report
ReportProducer	new	Report	Report
Reportq	changed	Report	Replaced by Report
Res	changed	Resource-like synchronisation	Synchronisation
ResObserver	new	Observer	Observation
Resource	lost	Resource-like synchronisation	not transferred
Sched	lost	Scheduling	replaced by ExecutionList
ShortGermanTF1	lost	Time to string	not transferred
Simulation	new	Simulation	Encapsulation
SimulationObserver	new	Observer	Observation
Stackdir	lost	ODEM-internal	not transferred
Starter	lost	ODEM-internal	not transferred
StatisticManager	new	Statistic	Statistics
StatisticObject	new	Statistic	Statistics
Sum	changed	Statistic	Statistics
Tab	changed	Statistic	Statistics
Table	new	Report data table	Report
TableDefinition	new	Report table structure	Report
Tag	new	Trace	Trace
Tally	changed	Statistic	Statistics
Timer	lost	Time events	not transferred
Trace	changed	Trace	Trace
TraceClient	lost	Trace	replaced by TraceConsumer
TraceConsumer	new	Trace	Trace
TraceFilter	new	Trace	Trace
TraceProducer	new	Trace	Trace
TraceServer	lost	Trace	replaced by Trace
TypedObject	new	C++ RTTI	Interface to standard C++ RTTI

Unifrom	changed	Random number generator	Random
utTableDef	new	Report	Report
Version	new	ODEMx version information	no comment
Wait	new	Synchronisation with child-processes	Synchronisation
Waitq	changed	Master-Slave synchronisation	Synchronisation
WaitqObserver	new	Observer	Observation
Class	**Status**	**Description**	**Comment**

New features

Observation

ODEMx introduces a system for observing individual objects. Observable objects provide an interface with call-back-functions for the events of that object. Observers implement this interface to handle selected events. An observable object inherits an observer management from the template class Observable. Observable objects also have an optional constructor parameter, which allows registering an initial observer.

Documentation:
`odemx::Observable`
`odemx::ProcessObserver`

Encapsulation

In contrast to ODEM, ODEMx does encapsulate simulations. While in ODEM the main program is always made a part of the simulation, ODEMx separates the simulation from its environment. Simulation specific data and services are put into the class Simulation instead of being placed in the global scope. For many objects of ODEMx, this requires a link to the Simulation class, which is in general handed over during construction.
The separation is often realised by the introduction of context-objects. Simulation for instance is the context-object for processes. Due to multiple inheritance Simulation also serves as a context-object for other types of objects, like random number generators, labelled objects and trace producers.
The advantage of the encapsulation is that a simulation can be programmed as an independent component without side effects on its environment. It is for instance possible to run a simulation in a simulation.

Documentation:
`odemx::Simulation`
`odemx::Process`

Coroutines

ODEMx separates the implementation of coroutines from the implementation of processes for process-simulation. ODEM on the other hand implements the coroutine functionality inside its process implementation. The coroutine implementation of ODEMx is apart from a few concepts in the Utilities module independent from the rest of ODEMx. It is designed to be portable and already ported to Windows higher than Windows 95 (x86 architecture) as well as Unix and Linux (Sparc, and x86 architecture) though in some compiler-os-platform configurations there could be problems.
The techniques used to realise coroutines are based on ODEM, which in turn has used previous works from HANSEN and others. ODEMx however introduces the encapsulation of coroutines in separate contexts.

Process queue

While in ODEM processes are designed to be part of a linked list, ODEMx uses STL containers to manage object collections. `ProcessQueue` is introduced in ODEMx to manage such collections. `ProcessQueue` allows different sorting schemes like considering the priority or the execution time of a process. Derived from `ProcessQueue` is the class `Queue`, which is used by synchronisation objects like `Res`, `Bin`, and `Waitq`.

Lost features

Memo

In ODEM, many classes include the functionality to be part of a linked list. The `Memo` concept of ODEM uses this property to provide a synchronisation technique. A `Process` object can wait in a `Memo` object until, the `Memo` object is 'available'. From `Memo` derived classes can override a function to redefine the 'available' condition. `Process` from ODEM for instance is also a `Memo` object, which redefines 'available'. A Process in ODEM is 'available' if it is terminated.
ODEMx does not include the Memo feature. It is however possible to realise `Memo`-like synchronisation with the Observation feature. The ODEMx class `Wait` in the module Synchronisation is an example for this.

Port

In the HU-Release of ODEMx, there is no concept for a buffered process-communication. A user will have to develop its own class for this purpose. ODEM had such a concept in its `Port` mechanism.

Time translation

In the later versions of ODEM, a time to string to time translation was introduced to allow human readable time strings other than simple floating point numbers. This feature was introduced for a special application of ODEM. ODEMx has not yet such a feature, but is likely to get a similar one in a future release.

OpenGL visualisation

ODEM had an experimental 3D visualisation of simulation events. Experiences from this 3D-project had lead to changes in the trace-mechanism, which finally concluded the current Trace concept of ODEMx. A visualisation might be available in ODEMx in a future version. Nevertheless, the old 3D-Trace of ODEM is not transferable without a redesign.

Continuous Graph

ODEM provides the class `Graph` which logs the state changes of a `Continuous` object in a text format used by external visualisation tools. A similar tool is already present in ODEMx (`odemx::ContuTrace`). The current version however does no generate output in the exact text-format like `Graph`.

Buffer

Like the missing `Port` the simplified `Buffer` system is not transferred to ODEMx.

Empirical

Empirical from ODEM is not available in ODEMx.

Changes

Synchronisation

ODEMx provides the following synchronisation components: `odemx::Res`, `odemx::Bin`, `odemx::Waitq`, `odemx::Condq` and `odemx::Wait`. The first four components are transferred from ODEM but are changed. `Res` and `Bin` for instance are not derived from a common base class `Resource`. The constructors of both classes have additional parameters compared to `Res` and `Bin` from ODEM, which reflect the Encapsulation and the Observation feature. Both classes have more `get*` methods to receive information about their statistic. Both classes have a report function according to the Report changes. The basic synchronisation functions however have not changed.
Similar things can be said about `Waitq` and `Condq`. The general meaning has not changed while the classes have been adjusted to reflect the ODEMx features. In `Waitq` and `Condq` however, small changes have been applied to the classic synchronisation functions as well.

Documentation:
`odemx::Res`
`odemx::Bin`
`odemx::Waitq`

odemx::Condq

Statistics

Many of the statistic components from ODEM have been transferred to ODEMx. The components in ODEMx share the update() function with their predecessors as well as the used algorithms. They were also changed to match the Encapsulation and the Report feature. Furthermore, several get* methods for the statistical data were added.

Documentation:
odemx::Accum
odemx::Count
odemx::Histo
odemx::Regress
odemx::Sum
odemx::Tally

Random

The random number generators (RNG) in ODEMx were taken from ODEM and changed to match the features of ODEMx. They have additional get* methods for statistical information, and require a pointer to a DistContext object during construction. They also provide a report function for the Report feature of ODEMx.
The DistContext class was introduced to support the Encapsulation feature. All random number generators linked to one DistContext are independent from each other, while the RNG in different DistContext produce the same sequence of numbers (unless the seed was changed manually).

Documentation:
odemx::Draw
odemx::Erlang
odemx::Iconst
odemx::Negexp
odemx::Normal
odemx::Poisson
odemx::Randint
odemx::Rconst
odemx::Uniform

Process

The Process class of ODEMx replaces the Process class and the Discrete class of ODEM. Its interface is more like the interface of Discrete than of Process. The following table matches the different scheduling functions of Discrete to the functions of ODEMx Process:

Discrete	**odemx::Process**
start(NOW)	hold()
start(AT, t)	holdUntil(t)
start(AT, t, PRIOR)	activateAt(t)
start(DELAY, t)	holdFor(t)
start(DELAY, t, PRIOR)	activateIn(t)
activate(NOW)	hold() or activate() (FIFO or LIFO)
activate(AT, t)	holdUntil(t)
activate(AT, t, PRIOR)	activateAt(t)
activate(DELAY, t)	holdFor(t)
activate(DELAY, t, PRIOR)	activateIn(t)
activate(BEFORE, q)	activateBefore(q)
activate(AFTER, q)	activateAfter(q)
hold(t)	holdFor(t)
passivate()	sleep()
e_interrupt()	interrupt() (!see documentation!)
cancel()	cancel()

ODEMx `Process` has only one priority attribute, which is used in scheduling as well as in synchronisation queues. The interrupt function in ODEMx has a different effect than the `e_interrupt` of Discrete. In ODEMx an interrupt causes an immediate activation (`activate()`). The interrupted process is responsible to handle the interrupt.

Documentation:
`odemx::Process`

Continuous

The `Continuous` class of ODEMx is derived from `Process`. The algorithm used to compute the state changes is taken from ODEM. Some of the `Continuous` functions in ODEM are also transferred to ODEMx. Other than the ODEM version of Continuous, the ODEMx version does not log the state changes on its own. If a user wants this service, he/she has to use the ODEMx class `ContuTrace`.

Documentation:
`odemx::Continuous`
`odemx::ContuTrace`

Object labels

The support of object labels has changed in ODEMx. It is now provided by the classes `LabeledObject`, `LabelScope` and `DefLabeledObject`. In contrast to ODEM labels are no longer unique to the program but to a certain `LabelScope`. Each simulation has of course its own scope.

Documentation:
`odemx::LabeledObject`
`odemx::LabelScope`
`odemx::DefLabeledObject`

Trace

ODEMx introduces the class `HtmlTrace`. Objects of this class generate Html output from simulation events. `HtmlTrace` also provides a simple filter for events to reduce the generated output. The trace control is implemented in the ODEMx `Trace` class, which is a base class of Simulation.

Documentation:
`odemx::HtmlTrace`

Report

ODEMx supports several reports in one simulation. A report is generated by an object of the `HtmlReport` class. Other than with ODEM, a user has to register model components to a report. The final report-generation is triggered manually by the function call `generateReport()`.

Documentation:
`odemx::HtmlReport`

Examples

Base1.cpp

Basic simulation techniques of ODEMx are introduced in this example.

A process simulation contains a number of processes that describe the active elements in a model. A process contains a sequence of actions. These actions are timeless. Time consumption is triggered with special time-operations. In this example, several time-operations are presented together with an example process.

The first Process is a simple timer. It does write a '`.`' every full (1.0) tic in time. We label all instances of this class '`TimerA`'. ODEMx changes the label on its own to prevent confusion. It will add a number to the label if more than one Timer is created in the simulation. This procedure guarantees that every label is unique in a simulation. When we create a Timer object we will have to provide a pointer to the `Simulation` we want it to participate.

```
#include <odemx/base/Process.h>
#include <iostream>
using namespace std;
using namespace odemx;

class TimerA : public Process {
public:

TimerA(Simulation* sim) : Process(sim, "TimerA") {}
```

The behaviour of a user process is defined by the implementation of the pure virtual function `int main()`. The return value of this function is stored and can be accessed with `getReturnValue()`. As long as `main()` hasn't finished, the return value is not valid. You can use `hasReturned()` to check this condition.
We use `holdFor()` to let the time pass. `holdFor()` requires a period. We could also use `activateIn()`. The difference between these two is seen if more than one process is scheduled at the same time. `holdFor` would than schedule the current process as the last while `activateIn` would schedule the current process as the first one.

```
virtual int main()
{
    while (true)
    {
          holdFor(1.0);
          cout << '.';
    }
    return 0;
}

}; // End of class TimerA
```

For this example, we can use `DefaultSimulation`.. In applications that are more complex, it is recommended to provide a custom-made simulation class.

```
int main(int argc, char* argv[])
{
TimerA* myTimer=new TimerA(getDefaultSimulation());
```

The new process is just created by now. It won't be executed because it is not activated. A process can be activated by any of these methods: `hold()`, `holdFor()`, `holdUntil()`, `activate()`, `activateIn()`,

acitvateAt(). hold() and activate() are equal to holdFor(0.0) and activateIn(0.0). holdUntil(t) and activateAt(t) are equal to holdFor(t-now) and activateIn(t-now).

```
myTimer->activate();
```

There are three ways to compute a simulation. Firstly, you can run() the simulation until it is finished. A simulation is finished if there is no active process left or it is stopped with exitSimulation(). Secondly, you can compute a simulation step() by step. Thirdly, you can run a simulation until a given time is reached or passed with runUntil(). Because the process TimerA is running forever, we shouldn't use run(). Instead we try both step() and runUntil().

```
cout << "Basic Simulation Example" << endl;
cout << "========================" << endl;
for (int i=1; i<5; ++i)
{
    getDefaultSimulation()->step();
    cout << endl << i << ". step time=";
    cout << getDefaultSimulation()->getTime() << endl;
}

cout << endl;
cout << "continue until SimTime 13.0 is reached or passed:";
getDefaultSimulation()->runUntil(13.0);
cout << endl  << "time=" << getDefaultSimulation()->getTime() << endl;
cout << "========================" << endl;
return 0;

} // End of main
```

After compilation, this example produces the following output:

```
Basic Simulation Example
========================

1. step time=0
.
2. step time=1
.
3. step time=2
.
4. step time=3

continue until SimTime 13.0 is reached or passed:.........
time=13
========================
Press any key to continue
```

continuousExample.cpp

The basic continuous simulation techniques of ODEMx are introduced in this example.

Apart from discrete processes, a simulation can also contain so called continuous processes. A discrete process does change the system-state only at discrete moments in simulation time with the actions defined in its main() function. The progress of time is triggered with functions like holdFor, activateAt and so on. See the previous example for details. A continuous process instead changes the state of a system continuously. Such a process could be for example the melting of a block of metal inside an oven. At every time, during the melting process, the temperature of the metal is changed. Of course, real continuity is quite impossible in the discrete world of our computers. Therefore, ODEMx has to approximate continuous state changes with a step by step computation.

The first continuous process is very simple. It only defines a sinus/co-sinus oscillator. The construction of a continuous object is a little different to that of a Process object. You also have to provide the number of state-variables used by your process. In this case, we need two variables.

```
#include <odemx/base/Continuous.h>
#include <cmath>
using namespace odemx;

class Oscillator : public Continuous {
public:
Oscillator() : Continuous(getDefaultSimulation(), "Oscillator", 2) {};
```

The `main()` function of a continuous process is quite similar to that of a discrete process. You can do everything that is possible in `Process`. That's why a continuous can behave just like a discrete process. However, it can also go through phases of continuous state changes.
Before you can start the solver, which is computing the continuous state changes, you will have to initialise the state variables. Than you should set some parameters for the internal solver. These parameters include error sensitivity and the step length.
The step length is set with `setStepLength()`. The first parameter sets the minimal step length, while the second defines the maximal step length. The internal solver will compute new states in steps. Between every step, the process holds. The actual length of each step will depend on numerical errors, peer processes, state events, and the parameters provided with `setStepLength()`. However, the actual step length will not exceed your provided maximum. If the step length has to be reduced because of numerical errors or state events, it will not be reduced below your provided minimum.
The error sensitivity is set with `setErrorlimit()`. The first parameter defines whether the errors should be measured relative to the value of the state variables (0) or absolute (1). The second parameter sets the maximum error acceptable (relative or absolute). If the actual error exceeds this value, the solver will try to reduce the step length. If this fails, because the step length is already to small, a simulation error is reported.
Finally, the continuous phase is started with the function `integrate()`. It is stopped either by a time event, a state event or an interrupt from another process. The time event is set by the first parameter. If it is zero, the solver will run forever. Otherwise, it will run to the given absolute time. If the provided time has already passed it will return at once. The return-value of `integrate()` will be `0` if the time event was hit and 2 if the process was interrupted.

```
protected:
virtual int main()
{
    state[0]=1;
    state[1]=0;

    setStepLength(0.01, 0.1);
    setErrorlimit(0, 0.1);


    integrate(20.0, 0);
    return 0;
}
```

Every Continuous process has to provide a specific `derivatives()` function. In this function you define how the state is changed during the time. You do this by setting the change-rate for every state variable. Although you don't have to, you can include the time, provided by the parameter `t`, in your computation. However, never use `getCurrentTime()` to get the time.

```
virtual void derivatives (double t)
{
    rate[0]=-state[1];
    rate[1]=state[0];
}
}; // End of class Oscillator
```

The `FreeFall` continuous process needs only one state variable. It demonstrates the use of parameter t in `derivatives()`. The result is an idealistic free fall.

```
class FreeFall : public Continuous {
public:
FreeFall() : Continuous(getDefaultSimulation(), "FreeFall", 1) {};

protected:

virtual int main() {
    state[0]=0.0;
    setStepLength(.1,1);
    integrate(20.0, 0);
    return 0;
}
```

Again, never use `getCurrentTime()` to include the current time in your computation. The reason for this is, that `derivatives()` is called multiple times for each step and these calls are not synchronised to the official time in the simulation.

```
virtual void derivatives (double t) {
    double g=9.81;
    rate[0]=t*g;
}

}; // End of class FreeFall
```

`RealFall` simulates a 'real fall', which is slowed down by friction. As well as in `FreeFall`, we don't set the error limits. We can do so because ODEMx has default settings for error limits and step length. The default error limit is a relative (0) error of 0,1.

```
class RealFall : public Continuous {
public:
RealFall() : Continuous(getDefaultSimulation(), "RealFall", 2) {};

protected:
virtual int main()
{
    state[0]=2.0;
    state[1]=0.0;
    setStepLength(.1, 1);
    integrate(20.0, 0);
    return 0;
}

virtual void derivatives (double t) {
    double k=0.5;
    double g=-9.81;
    rate[0]=state[1];
    rate[1]=g - k*state[1];
}

}; // End of class RealFall
```

`RealBounce` finally demonstrates the use of state events. The function `hitGround()` is used to check a state event. The signature of functions that can be used as state-event-functions is `bool(Process::*)()`. ODEMx uses pointer to member functions if it needs a call-back. The advantage is, you can use member functions with

full access to internal data of your classes to check state events. The cost of this design decision is that you always have to cast the address of your state-event-function to the type `Condition`. A state function has to return true if a state event has occurred. Consider that the computation of state changes is done in steps. Because of that, it is very unlikely to hit a state event exactly. This has to be taken into account for the programming of a state-event-function. If a state event has occurred (or passed) the internal solver starts a binary search to get closer to the exact event time. Finally, if it is close enough (minimum step length) the computation is stopped and `integrate()` returns `1`.

```
class RealBounce : public Continuous {
public:
RealBounce() : Continuous(getDefaultSimulation(), "RealBounce", 3) {};

bool hitGround()
{
    return state[0]<=0.0;
}

protected:
virtual int main()
{
    double g=-9.81;
    state[0]=2.0;
    state[1]=0.0;
    state[2]=g;
    setStepLength(0.01, 0.1);
```

`RealBounce` uses the return value of `integrate()` to control the computation. Remember that `integrate()` returns `0` if a time event occurred, `2` if the process was interrupted and `1` if a state event was detected. `integrate()` is called with the time event 20.0 and the state-event-function `hitGround()`. If the state event is hit we reflect the movement '`state[1]=-(0.8*state[1])`' and continue until the simulation time exceeds 20.0.

```
    while (integrate(20.0, (Condition)&RealBounce::hitGround)==1)
    {
          if (fabs(state[1])<0.01 && state[0]<0.01)
          {
                //
                // We must stop the ball if it has lost too much energy.
                // Otherwise, we would produce an annoying loop because the
                // state event would stop integration in every step.
                //

                state[1]=0.0;
                state[2]=0.0;
          }
          state[1]=-(0.8*state[1]);
    }
    return 0;
}

virtual void derivatives (double t)
{
    double k=0.5;
    rate[0]=state[1];
    rate[1]=state[2] - k*fabs(state[1]);
    rate[2]=0.0;
}
```

```
}; // End of class RealBounce

int main(int argc, char* argv[])
{
    Oscillator osci;
    FreeFall free;
    RealFall real;
    RealBounce bounce;
```

To log the state changes we use the class ContuTrace. ContuTrace observes a provided continuous process and logs all state changes into a text file. The file is managed by ContuTrace. The name is either set in the constructor or build from the name of the observed continuous process. If you run the simulation you will find the four text files: Oscillator_trace.txt, FreeFall_trace.txt, RealFall_trace.txt, RealBounce_trace.txt.

```
    ContuTrace tracer[] =
        {  ContuTrace(&osci),
           ContuTrace(&free),
           ContuTrace(&real),
           ContuTrace(&bounce)};

    osci.activate();
    free.activate();
    real.activate();
    bounce.activate();

    //
    // We can run the simulation without a time limit because
    // the defined continuous processes end at 20.0 .
    //
    getDefaultSimulation()->run();
    return 0;
} // End of main
```