Remarks on the Implementation of the Modelica Standard Tables

Thomas Beutlich Gerd Kurzbach Uwe Schnabel
ITI GmbH
Schweriner Straße 1, 01067 Dresden, Germany
{beutlich, kurzbach, schnabel}@itisim.com

Abstract

This article reveals some implementation details regarding the C code of the revised table interpolation blocks released with the Modelica Standard Library (MSL) 3.2.1. The emphasis is placed on the unique features of the CombiTimeTable which are the discontinuities by time events and the periodic extrapolation. Basic information on the interpolation by Akima splines and the available table array memory optimization options are mentioned.

Keywords: univariate interpolation, bivariate interpolation, periodic extrapolation, time events, Akima splines

1 Introduction

For many years there has been no (open source) implementation of the table interpolation blocks of the MSL. Thus, Modelica simulation tools either did not support the table blocks or needed to provide a custom implementation that could lead to different simulation results. One objective of releasing a backward compatible MSL 3.2.1 was to provide an open source implementation of the table blocks based on the Modelica external object interface. This table implementation was named *Modelica Standard Tables* and comprises the following four blocks for univariate and bivariate interpolation

- Modelica.Blocks.Sources.CombiTimeTable,
- Modelica.Blocks.Tables.CombiTable1D,
- Modelica.Blocks.Tables.CombiTable1Ds and
- Modelica.Blocks.Tables.CombiTable2D.

The C header and source files of the Modelica Standard Tables are publicly available from `https://svn.modelica.org/projects/Modelica/trunk/Modelica/Resources/C-Sources`. The C interface functions all start with prefix `ModelicaStandardTables_`. The constructors and destructors of the external table objects have suffix `_init` and `_close`, respectively. If the table data is to be read from an ASCII text file or a MATLAB MAT-File, an interface function with suffix `_read` is used, i.e. file I/O is not part of the construction of the external table object. The actual interpolation functions are labeled by trailing `_getValue` and `_getDerValue` for the interpolation function and the interpolated derivatives, respectively.

2 CombiTimeTable

The block Modelica.Blocks.Sources.CombiTimeTable is different from standard univariate interpolation since discontinuities (by time events) and periodic extrapolation are considered. For instance, periodic and discontinuous signals like saw-tooth or square-wave w.r.t. simulation time can be modeled in a very convenient and compact way.

2.1 Time events

Time events always occur at the table boundaries (t_{min} and t_{max}) of the sample points, i.e. if interpolation switches to extrapolation.

- In case of linear interpolation, additional time events can be modeled by repetition of sample points in the table array. It is guaranteed that there are no time events at interval boundaries with a simple sample point only. Thus the time coordinates are not required to be strictly increasing but monotonically increasing.
- In case of interpolation by constant segments, every interval boundary (defined by the sample

points) leads to a time event, whether or not there is an actual discontinuity in the ordinate values. The time coordinates are not required to be strictly increasing but monotonically increasing. In fact, repeated sample points are not appropriate for the interpolation result.

- Additional time events are not possible in case of interpolation with Akima splines. The time coordinates are required to be strictly increasing. (Thus care must be taken when changing the interpolation smoothness if there are repeated sample points.)

The static function `isNearlyEqual` from source file ModelicaStandardTables is used to tell if two double-precision floating-point numbers are (nearly) equal with relative threshold `_EPSILON` (set to 10^{-10} by default).

The calculation of the next time event t_E is performed in the interface function `ModelicaStandardTables_CombiTimeTable_nextTimeEvent`. For numerical reasons related to the floating-point arithmetic used, the current time t is incremented by a small fraction of the time table length, that is `_EPSILON` $\cdot (t_{max} - t_{min}) > 0$. It is guaranteed that t_E is greater than the current time t and that no time events are missed if the distance between two consecutive time events is greater than this numerical increment.

If no future time event is found `_nextTimeEvent` returns `DBL_MAX`. Hence, care should be taken by a Modelica simulation tool to avoid floating-point overruns during the event handling.

For debugging purposes, time events can be traced (by means of `ModelicaFormatMessage`) by defining `DEBUG_TIME_EVENTS` where each message line corresponds to one time event. For instance, linear interpolation and extrapolation of the 4×2 example time table [0.25, 30; 0.5, 40; 0.5, 10; 0.75, 30] results in three time events (Fig. 1).

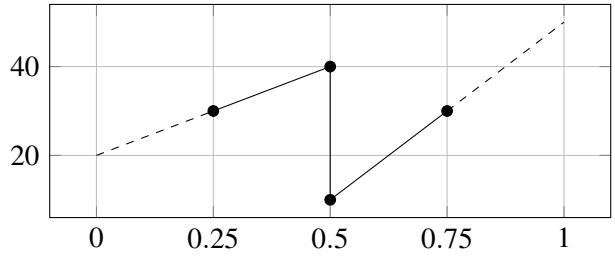

Figure 1: Linear interpolation of time table [0.25, 30; 0.5, 40; 0.5, 10; 0.75, 30] results in three time events.

The four messages (including the initial event) are

```
At time 0.00000: 1. time event at 0.25000
At time 0.25000: 2. time event at 0.50000
At time 0.50000: 3. time event at 0.75000
No more time events for time > 0.75000
```

In order to return the correct function values at the time events (i.e. during the event iterations), the interface functions `_getValue` and `_getDerValue` require not only the next event time but also its pre-value as input arguments.

2.2 Periodic extrapolation

For tables with periodic extrapolation, the numerically stable detection of periodic time events is rather tricky.

A first implementation was discarded as it turned out that the periodic time events were not reliably detected in all cases. The main reason is due to the IEEE 754 binary floating-point arithmetic where t and $t - n \cdot T$ cannot be used simultaneously to detect the exact location of an event interval or periodic time event. Here, t denotes the floating-point number (FPN) of the current time and $T = t_{max} - t_{min}$ the FPN of the period of the table. Variable n is a signed integer and denotes the multiple of the period. Furthermore, let t_E be the FPN of the event time (to be detected) and t_i the FPN of the corresponding event interval boundary time from the table. There are FPNs such that the inequality $t \geq t_E = t_i + n \cdot T$ is true, i.e. indicates a time event, but where a simple rearrangement of the inequality to the form $t - n \cdot T \geq t_i$ does not hold. This illustrates the fact that floating-point operations cannot be used to exactly evaluate floating-point comparisons, and therefore cannot be used to reliably detect periodic time events.

The final implementation is based on (nonnegative) integers

- `nEventsPerPeriod`, the number of time events per period and
- `eventInterval`, the (discrete) event interval marker.

During the very first call of function `_nextTimeEvent`, the number of time events per period is determined from the time coordinates of the table array. There is always a time event per period at the

DOI 10.3384/ECP14096893

table boundary, thus `nEventsPerPeriod` ≥ 1. Furthermore, the start and end indices of each of the so-called *event intervals* are determined and stored in the integer array `intervals`.

As an example, the 5×2 time table [0, 10; 0.25, 30; 0.5, 40; 0.5, 10; 0.75, 30] is considered.

- There are two time events per period in case of linear interpolation. The indices of the event intervals are [0, 2] and [3, 4], i.e. the first interval ranges from time 0 to time 0.5 and the second interval from 0.5 to 0.75 (Fig. 2).

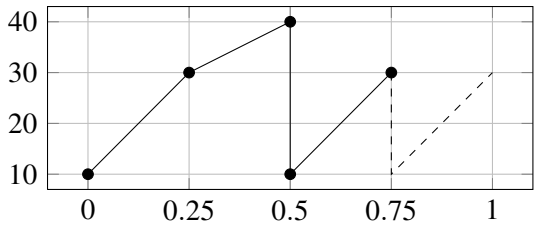

Figure 2: Two time events per period

- There are three time events per period in case of interpolation by constant segments. The indices of the event intervals are [0, 1], [1, 2] and [3, 4]. Repeated sample points are ignored since their ordinate values can never be taken by the interpolating function (Fig. 3)

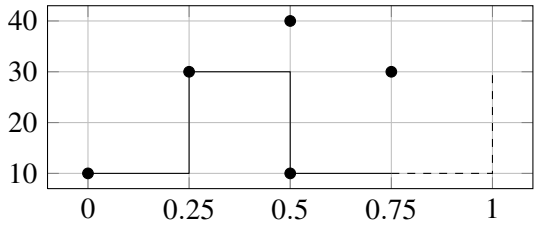

Figure 3: Three time events per period

The event interval marker `eventInterval` uses one-based indexing. It is properly initialized once in the very first call of function `_nextTimeEvent` since interpolation does not need to start in the first event interval by default. Along with both integer variables and the event intervals array, the initial offset time `tOffset` $= n \cdot T$ is determined and stored.

Each subsequent call of `_nextTimeEvent` now increments the event interval marker by one and resets it to 1 once it overruns, i.e. gets greater than the number of time events per period. This can be derived from the fact that the input variable (time) is usually increasing (w.r.t. time). Then it is known that there is exactly one time event per event interval. There is an event interval correction implemented in functions `_getValue` and `_getDerValue` that sets the time to be interpolated to either the left or right event interval value in case of floating-point inaccuracies. This implementation guarantees that no time events are missed and that the correct function values of the interpolating function at the event interval boundaries are returned.

3 Interpolation by Akima splines

Hiroshi Akima's original articles [1, 2] were used for the implementation of the univariate and bivariate interpolation by Akima splines. His reference implementation (in FORTRAN 77) was not used. Furthermore, the FORTRAN 90 code of SOSIE [3] from Laurent Brodeau was studied for an efficient calculation of the coefficients for bivariate splines.

3.1 Spline coefficients

There are different possibilities to calculate and store the polynomial spline coefficients of the interpolating function.

1. Pre-calculate all coefficients during the initialization and store them within the external table object.

2. Calculate the coefficients whenever needed (and possibly store only the last calculated set of coefficients if used for evaluating the partial derivatives in the same simulation time step).

3. Allocate enough memory during initialization to store all coefficients, calculate them whenever needed and store them whenever calculated. The advantage is that for very large tables where only a few different data parts are accessed there will be no superfluous calculation at all and initialization time is short. The disadvantages are the high memory usage and the unpredictable execution time of a simulation time step (as it is never known if the calculated coefficients are already available).

This implementation of the Modelica Standard Tables uses the first option where all required coefficients of the entire table array are calculated during initialization in function `spline1DInit` (for univariate Akima splines) and `spline2DInit` (for bivariate Akima splines).

3.2 Differentiability

The current univariate Akima spline interpolation always uses non-periodic table boundary conditions which may lead to a discontinuous interpolating function or derivative at the table boundaries for periodic extrapolation with the CombiTimeTable (Fig. 4).

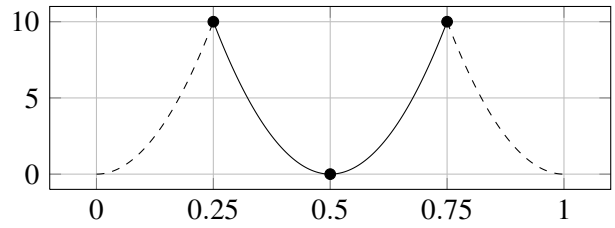

Figure 4: Periodic extrapolation of time table [0.25, 10; 0.5, 0; 0.75, 10] by Akima splines results in a interpolating function that is not (continuously) differentiable.

In all other cases it is guaranteed that the univariate or bivariate Akima spline interpolating function is continuously differentiable everywhere, especially at the table boundaries (Fig. 5).

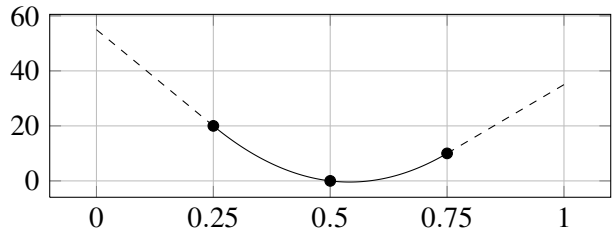

Figure 5: The boundary slopes of the Akima splines are used to linearly extrapolate the one-dimensional table [0.25, 20; 0.5, 0; 0.75, 10].

4 Array memory optimizations

Advanced array memory optimization features are implemented and explained below.

4.1 Shared table arrays

If multiple table objects refer to the same table array of the same file, this table array is read and stored in memory multiple times since external objects are mutually independent by default. In order to avoid superfluous file input access and to decrease the utilized memory there is a C++ implementation of a global table array management on top of the C implementation, guarded by predefined macro `__cplusplus`. The C or C++ compilation can be toggled by compiler flag `-x c` and `-x c++` for GCC or flag `/TC` and `/TP` for Microsoft Visual C++. In the case of a C++ compilation an additional static variable of type `TableShareMap` (using the `std::map` container from the STL) with reference counting is introduced. Write access of this global variable `tableShare` (e.g. insertion or erasure of tables) is thread-safe, i.e. guarded by a critical section (on Windows platforms) or pthread mutex (on Linux platforms) and implemented by `struct CriticalSectionHandler`.

A table array update is usually not required and therefore not implemented. Shared arrays of spline coefficients are also not implemented, i.e. each external table object always calculates and locally stores the spline coefficients it requires.

4.2 Shallow copy of table arrays

This optimization is only relevant if the table array is defined within the simulation model, i.e. if it is known at compile-time and not read from the file at simulation run-time. The complete table array memory that is passed from the simulation model to the constructor (the `_init` function) of the external table object is allocated and copied (“deep copy”). This is the safe default case as nothing can be assumed about the lifetime of the table array of the simulation model. Thus the table memory is (temporarily) held twice: locally within the external table object and at the outside model. If the outside table array is known to be constant and to have a longer simulation lifetime than the external table object, the deep array copy can be avoided by defining `NO_TABLE_COPY`. In this case, the outside table array memory is used within the external table object (“shallow copy” of the passed array pointer).

5 Conclusions

An open source implementation of the table blocks

- Modelica.Blocks.Sources.CombiTimeTable
- Modelica.Blocks.Tables.CombiTable1D
- Modelica.Blocks.Tables.CombiTable1Ds
- Modelica.Blocks.Tables.CombiTable2D

DOI 10.3384/ECP14096893

is available with the MSL 3.2.1 as of August 2013. The Modelica code is provided under Modelica License 2. The C/C++ code of files ModelicaStandardTables and ModelicaMatIO [4] is provided under the new BSD license. As a result, all parts of the MSL are now available in a free implementation.

Additionally it should be mentioned that this implementation added new features to the table blocks.

- The new option ConstantSegments was added for the Smoothness parameter.
- The new option NoExtrapolation was added for the Extrapolation parameter.
- The table outputs can be differentiated once (with exception of the potentially discontinuous ConstantSegments).
- All MATLAB MAT-File formats are supported by an adapted library libmatio [4] (provided by ModelicaMatIO header and source file). Whereas MAT-File formats v4 and v6 are supported without additional dependencies by libmatio, the v7 format requires the zlib [5] library and compilation with preprocessor macro `HAVE_ZLIB=1` and the v7.3 format requires the hdf5 [6] and szip [7] libraries and compilation with preprocessor macro `HAVE_HDF5=1`.
- The support of tables provided as static C arrays in the user header file usertab.h was revised. This is relevant for real-time systems without a file system and where the table data is known at compile-time.

Last but not least, 120 test models in ModelicaTest.Tables with reference results have been created.

6 Acknowledgement

The presented work was paid for by the Modelica Association.

Grateful acknowledgements go to

- Martin Otter for the first implementation (from 1997 till 2001) of the table interpolation blocks and for constructive discussions on the improvement of the new implementation,
- Hans Olsson and Martin Sjölund for valuable advice on the improvement of the implementation,
- Christopher C. Hulbert for sharing a MAT-File C library and providing a patch of `Mat_VarReadDataLinear` [4],
- David Zaslavsky for initiating an interp2d library compatible with the GNU Scientific Library (GSL) [8],
- John C. Beatty for sharing a simple utility to concatenate multiple C source files into a single C source file [9] that was used to merge header and source files of [4] to single files ModelicaMatIO.h and ModelicaMatIO.c,
- Laurent Brodeau for sharing a bivariate Akima algorithm (in FORTRAN 90) [3].

References

[1] Hiroshi Akima. A new method of interpolation and smooth curve fitting based on local procedures. *J. ACM*, 17(4):589–602, October 1970.

[2] Hiroshi Akima. A method of bivariate interpolation and smooth surface fitting based on local procedures. *Commun. ACM*, 17(1):18–20, January 1974.

[3] Laurent Brodeau. SOSIE is Only a Surface Interpolation Environment. `http://sosie.sourceforge.net`.

[4] Christopher C. Hulbert. MAT File I/O library version 1.5.2. `http://matio.sourceforge.net`.

[5] Jean-Loup Gailly and Mark Adler. A general purpose compression library version 1.2.8. `http://zlib.net`.

[6] The HDF Group. Hierarchical data format version 5. `http://www.hdfgroup.org/HDF5`.

[7] The HDF Group. Science data lossless compression library version 2.1. `http://www.hdfgroup.org/doc_resource/SZIP`.

[8] David Zaslavsky. A 2D interpolation library compatible with GSL. `https://github.com/diazona/interp2d`.

[9] John C. Beatty. A very simple inclusion processor. `https://www.student.cs.uwaterloo.ca/~cs241/cLibs/mergeCsource`.