# *Mathematica* Connectivity to Interval Libraries filib++ and C-XSC*

Evgenija D. Popova

Institute of Mathematics and Informatics
Bulgarian Academy of Sciences
Acad. G. Bonchev str., block 8, 1113 Sofia, Bulgaria
epopova@bio.bas.bg

**Abstract.** Building interval software interoperability can be a good solution when re-using high-quality legacy code or when accessing functionalities unavailable natively in one of the software packages. In this work we present the integration of programs based on the interval libraries `filib++` and C-XSC into *Mathematica* via *MathLink* communication protocol. On some small easily readable programs we demonstrate: i) some details of *MathLink* technology, ii) the transparency of numerical data communication without any conversion, iii) the advantage of symbolic manipulation interfaces — the access to the external compiled language functionality from within *Mathematica* is often even more convenient than from its own native environment.

**Keywords:** Software interoperability, interfacing, interval software, C-XSC, `filib++`, *MathLink*, *Mathematica*, external programs.

## 1 Introduction

As the numerical methods based on interval analysis expand in their range and applications, the number and diversity of interval software increase rapidly. The existing interval software ranges from libraries for application development to fully interactive software systems [8]. However there are only a few interval software comparing studies. They were hampered by the diversity in the implementation supporting environments and in the interval data representations (see e.g. [2] and the references given therein). The provided functionality also varies from fairly basic and general to highly specialized. Although some specialized methods are brought to reliable, high-quality and fast implementations, they remain isolated software systems. Some specific software tools are built on the top of other more general interval libraries but there is no single environment supporting all (or most) of the available interval methods. Many problem solving routines require symbolic or structured input data and building corresponding application programming interfaces would facilitate their usage. On the other hand, most recent interval applications require a combination of diverse

* This work was partially supported by the Bulgarian National Science Fund.

A. Cuyt et al. (Eds.): Numerical Validation, LNCS 5492, pp. 117–132, 2009.

methods. It is difficult for the end-users to combine and manage the diversity of interval software tools, packages, and research codes, even the latter being accessible. Two recent initiatives: [3], directed toward developing of a comprehensive full-featured library of validated routines, and [11], intending to provide a general service framework for validated computing in heterogeneous environment, reflect the realized necessity for an integration of the available methods and software tools.

It is commonly understood that quality comprehensive libraries are not compiled by a single person or small group of people over a short time [3]. Therefore, in this paper we present an alternative approach based on interval software interoperability. In Section 2 we discuss some aspects concerning this approach and outline its advantages. Since on the one hand the general-purpose environments for scientific/technical computing like Matlab, *Mathematica*, Maple, etc. possess several features not attributable to the compiled languages and on the other hand most of the interval software is developed in some compiled language for efficiency reasons, it is interesting to study the possibilities for interoperability between these two kinds of interval supporting environments. In this paper we focus on the interaction between *Mathematica* [17] and two external C++ libraries for interval computations, `filib++` [9], [10] and C-XSC [4], [7] via *MathLink* communication protocol [5], [15]. In particular, we present how to call external programs based on either of these two interval libraries from within a *Mathematica* session, thus exchanging data without using intermediate files but under dynamics and interactivity in the communication. The goal is to demonstrate some advantages of interval software interoperability. Namely, expanded functionality for both environments, symbolic manipulation interfaces for the compiled language interval software which often make access to the external functionality from within *Mathematica* more convenient even than from its own native environment. Section 3 presents some basics from *MathLink* technology for building external *MathLink* -compatible programs. The technology will be further demonstrated in more details in the next sections on some small and easily readable sample programs. This work does not intend to provide a complete *Mathematica* interface for the C-XSC and `filib++` interval libraries but to demonstrate the connectivity technology and some important specific aspects of the interaction between *Mathematica* and `filib++` in Section 4, or C-XSC in Section 5. The electronic supplementary archives that accompany this paper give: i) more insight into *MathLink* technology, ii) a framework for its expansion on other problems, and iii) more illustrations of the presented connectivity.

## 2 Some Aspects of Interval Software Interoperability

Compiling a library of full-featured, high quality, portable and uniform interval-based tools, as presented in [3], is an ambitious goal, requiring the work of many people over many years. In contrast to this approach, providing interoperability between the existing interval software may achieve similar goals at a considerably

lower price and development time. The following advantages could be achieved with respect to the development process.

- When building software interoperability the usual tedious and error-prone work is removed providing in the same time a safeguard against the re-implementation bugs.
- Since we are usually interested to connect software which is already brought to a high quality and efficiency, only the connectivity and interoperability need to be tested but not the connected software components. The team can concentrate on the overall concept and the new implementations.
- Thus, in some cases providing interval software connectivity and interoperability would require considerably less development time than building everything from scratch. Software interoperability, instead of re-implementation, is especially suitable for complicated methods and large software systems which are already brought to a high quality and efficiency.

In the same time, a connectivity between two (or more) interval software environments would provide: expanded functionality, compatibility of interval representations, increased possibility for comparison and testing, accessibility by a wider range of users, performance improvement when a compiled code is integrated into an interpretative environment. A possible drawback is that each of the software ingredients should be maintained and ported to different platforms separately. In general, the problems that may arise in providing interoperability depend on the software that is to be connected and the purpose of the interaction. For example, the details in connecting C and Fortran programs will depend on the particular Fortran and C compilers (their calling conventions) and the types of the parameters to pass.

It is well-known that the general-purpose and multi-platform computing environments like Matlab, *Mathematica*, Maple, etc. possess several features not attributable to the compiled languages: dynamics and interactivity of the environment, symbolic and algebraic computations, numerics on all data types, powerful graphics programming, interfaces and connectivity, etc. The killer applications of these also called computer algebra environments are symbolic manipulation, education and prototyping. Thus, the interoperability between a computer algebra system and external compiled language software would bring additional benefits. The former will get expanded functionality and increased performance, while the external software could benefit from symbolic manipulations, powerful graphics capabilities, suitable interfaces, etc. Although all Matlab, *Mathematica* and Maple support interval computations, due to efficiency reasons most of the interval software is implemented in some compiled language. Therefore, it is interesting to study the interoperability between these general-purpose environments and the interval software developed in compiled languages.

There are two basic forms of communication between two software systems: structured and unstructured. Unstructured communication is based on file reading and writing operations to exchange ordinary text. This simplest form of communication between two software systems has some important drawbacks with respect to interval software systems. Namely, the necessity of avoiding inevitable

input/output roundoff errors. For an example of such unstructured communication and the problems that have to be solved refer to [2]. Here we explore and demonstrate interval software interoperability via communication protocols. The idea of structured communication is to transfer data without using intermediate files, communicating with external programs on a higher level and exchanging more structured data or complete expressions with the external programs which are specially set up to handle such objects.

## 3 *MathLink* Basics

Extensively used within the *Mathematica* system itself, *MathLink* is *Mathematica*'s unique high-level symbolic interface standard for interprogram communication [5], [15]–[17]. With convenient bindings for a variety of languages, *MathLink* allows arbitrary symbolic objects — representing data, programs, or any other construct — to be efficiently exchanged between programs, on one computer or across a heterogeneous network. In this work we demonstrate one of the most common uses of *MathLink*: to allow external functions written in some compiled language to be called from within the *Mathematica* environment.

Given a function defined in an external program, then what is necessary to do in order to make it possible to call the function from within *Mathematica* is to add appropriate *MathLink* code that passes arguments to the function, and takes back the results it produces. The overall process consists of four steps:

1. Create an appropriate *MathLink* template for each external function;
2. Combine the template with the external source code into a communication module;
3. Process the*MathLink* template information and compile all the source code;
4. Install the binary in the current *Mathematica* session.

The intention is that the developers take pre-existing routines and with as little effort as possible (ideally with no source code changes to the routines themselves), package them so they can be called from *Mathematica*.

A *MathLink* template involves the following mandatory elements:

| | |
|---|---|
| :Begin: | begin the template for a particular function |
| :Function: | the name of the function in the external program |
| :Pattern: | the *Mathematica* pattern to be defined to call the function |
| :Arguments: | the arguments to the function |
| :ArgumentTypes: | the types of the arguments to the function |
| :ReturnType: | the type of the value returned by the function |
| :End: | end the template for a particular function |

*MathLink* templates are conventionally put in files with names of the form `file.tm`. Such files can also contain C source code, interspersed between the templates for different functions. When a *MathLink* template file is processed, two basic things are done. First, the `:Pattern:` and `:Arguments:` specifications are used to generate a *Mathematica* definition that calls an external function

via *MathLink*. Second, the `:Function:`, `:ArgumentTypes:` and `:ReturnType:` specifications are used to generate C source code that calls the desired function within the external program. Both the `:Pattern:` and `:Arguments:` specifications in a *MathLink* template can be any *Mathematica* expressions. Whatever is given as the `:Arguments:` specification will be evaluated every time the external function is called. The result of the evaluation will be used as the list of arguments to pass to the function.

Sometimes it may be necessary to set up *Mathematica* expressions that should be evaluated not when an external function is called, but instead only when the external function is first installed. This can be done by inserting `:Evaluate:` specifications in the corresponding *MathLink* template. Usually, an usage message and/or error messages for the *Mathematica* functions are defined after `:Evaluate:`. When an external program is installed, the specifications in its *MathLink* template file are used in the order they were given. This means that any expressions given in `:Evaluate:` specifications that appear before `:Begin:` will have been evaluated before definitions for the external function are set up.

Once a *MathLink* template for a particular external function is constructed, this template has to be combined with the actual source code for the function. By the *MathLink* communication protocol the external programs send and receive *Mathematica* expressions using the fundamental C data types. Therefore, if the source code is written in the C programming language, all that should be done is just adding a line to include the standard *MathLink* header file, and then inserting a small main program. The form of main required on different systems may be slightly different, the appropriate form is given in the *MathLink* Developer Kit [17] for every particular computer system.

Once the couple of appropriate template file and C/C++ source files that make *MathLink* function calls is set up, they should be processed to build a *MathLink*-compatible program. The template file must first be processed into a C source file using a program named `mprep` included in the *MathLink* Developer Kit. `mprep` converts template entries into C functions, passes other text through unmodified, and writes out additional C functions that implement a remote procedure call mechanism using *MathLink*. The result is a C source file `file.tm.c` that is ready for compilation. All source files must be compiled and then the resulting object code must be linked with the `libML.a` library and any of the other standard libraries required by the applications. `mcc` is a script that preprocesses and compiles *MathLink* source files. For more information on how to compile and run *MathLink* programs written in C on Unix systems see [16].

Finally, the `Install` function is used to launch a *MathLink*-compatible program and to make its functions available in a *Mathematica* session.

## 4 *MathLink* Connection to `filib++`

The library `filib++` is known as an efficient portable C++ library supporting interval arithmetic and fast computation of guaranteed bounds for interval versions of a comprehensive set of elementary functions [9], [10]. There are two specific

features of the library regarding the representation of real intervals. The standard input/output operators are overloaded for intervals but without appropriate directed rounding from/to the external decimal string format. Analogously, the interval constructors do not provide enclosure for not exactly representable data. This feature requires special considerations and data handling if the conversion errors have to be bounded. In this section we show that this problem can be transparently solved by calling the `filib++` functions from *Mathematica* via the *MathLink* communication protocol. The second feature of `filib++` is that the library can be used in two modes: normal and extended. In the normal mode, when an interval evaluation is not defined, an exception handling is activated which terminates the program with an error message. In the extended mode, intervals are defined over the set of floating-point numbers extended by $\{-\infty, +\infty\}$. To cope with the closed set of real numbers and to allow exception free computations, the designers of `filib++` have accepted the IEEE representation of $-\infty$ and $+\infty$, allowing the latter to be used as left or right bound of an extended interval. The empty interval is represented as `[NaN, NaN]`.

Let us consider a small *MathLink*-compatible external program which uses the extended interval data type of the library `filib++` to perform containment computations. Below is the source file `filibStdF.cpp` involving the C function `filibStdF()` evaluating a standard mathematical function over an interval.

```
#include "mathlink.h"            // MathLink include
#include <interval/interval.hpp> // filib include
typedef filib::interval<double,            /* simplify instantiation */
    filib::native_switched, filib::i_mode_extended> interval;
using namespace std;

void filibStdF(int fcode, double *data, long len)
{
  interval  x(data[0], data[1]), res;
  switch(fcode)
  { case  1: { res = acos(x); break; }
    case  2: { res = acosh(x); break; }
...........    not presenting all cases for simplicity
    case 28: { res = tanh(x); break; }
  }
// Initializing C variables passing the computed result to Mathematica
  double res_data[4];
  res_data[0] = x.inf();     // showing that the interval constructor
  res_data[1] = x.sup();     // does not perform any rounding
  res_data[2] = res.inf();
  res_data[3] = res.sup();

  MLPutRealList(stdlink, res_data, 4); //Send result back to Mathematica
  return ;
}
int main(int argc, char* argv[ ]) // Standard MathLink main function
{   return MLMain(argc, argv);  }
```

Here is the corresponding template file `filibStdF.tm` which must have the same name as the source file:

```
:Evaluate: filibStdF::usage = "filibStdF[fun, arg] uses the external
interval library filib++ to evaluate a standard mathematical function
'fun' of one argument over a specified real interval 'arg'."
:Begin:
:Function:          filibStdF
:Pattern:           filibStdF[fun_Symbol, arg_Interval]
:Arguments:         {fun/.{ArcCos->1, ArcCosh->2, ArcCot->3, ArcCoth->4,
                      .....  Tan->27, Tanh->28}, arg[[1]]}
:ArgumentTypes:     {Integer, RealList}
:ReturnType:         Manual
:End:
```

The `:Function:` line specifies the name of the C routine. The `:Pattern:` line shows how the routine will be called from within *Mathematica*. The names of the two routines do not have to be identical as in this case. Note that the arguments in the *Mathematica* function pattern are restricted: the first one to be a symbolic name and the second to be the *Mathematica* object `Interval`. The `:Arguments:` line specifies the expressions to be passed to the external program. In our case these expressions are not the same as the variable names on the `:Pattern:` line. The `:Arguments:` specification will be evaluated every time the external function is called. During the evaluation the first argument `fun` of the calling *Mathematica* function will be transformed into an integer number[1] and from the second argument `arg` only the first part, presenting a list of the interval end-points, will be taken. The result of the evaluation will be used as the list of arguments to pass to the C function. The `:ArgumentTypes:` and `:ReturnType:` lines contain special keywords used by `mprep` to create the appropriate *MathLink* function calls that transfer data across the link. There are six keywords (`Integer, Real, IntegerList, RealList, String, Symbol`) for some more common types of data. For example, the keyword `Integer` on the `:ArgumentTypes:` causes `mprep` to create a call to *MathLink* function `MLGetInteger` which transfers C `int`s. Using keywords `IntegerList` or `RealList`, however, an extra argument has to be included in the corresponding C function to represent the length of the list, see the declaration of `filibStdF()` above. If the external function needs to receive or return expression types that are not among the set handled automatically by `mprep`, or if the function returns different types of results (such as an integer or the symbol `$Failed`) in different situations, then the keyword `Manual` can be included on the `:ArgumentTypes:` lines to inform `mprep` that we will write our own calls to get the arguments or (as in our case) on the `:ReturnType:` line to put the results ourselves.

The external programs should not modify the arrays generated by the *MathLink* functions `MLGetRealList(), MLGetRealArray()`, etc. Since the external libraries we connect to *Mathematica* use special data types for representing intervals, **the main purpose of the communication modules that have**

[1] For simplicity, not all elements of the transformation rule are presented.

**to be developed is to initialize new variables having the corresponding specific data types with the incoming data, and after the actual computations to transform the computed results into variables of fundamental C data types that will be passed back to *Mathematica*.**

Once the `filibStdF.tm` and `filibStdF.cpp` files have been processed and compiled to produce an executable file called `filibStdF`, we can install it into a *Mathematica* session. The *Mathematica* commands, presented below, illustrate its usage.

```
In[1] := lnk = Install["filibStdF"]
Out[1] = LinkObject[./filibStdF, 2, 2]
```

The `Install` function launches the program and opens a link through which the external function can be called. The program sends to *Mathematica* the definitions for its functions specified in the template file along with whatever code is given on the `:Evaluate:` lines. Let us see our external function in action and request the evaluation of $\cos(x)$ over the interval $[-1, 3]$. The output wrapper function `InputForm` is used to show all the digits of the results.

```
In[2] := filibRes= filibStdF[Cos, Interval[{-1, 3}]]
         filibRes // InputForm
Out[2] = {-1., 3., -0.989992, 1.}
Out[3]//InputForm = {-1., 3., -0.9899924966004472, 1.}
```

In order that the user can monitor the input/output communication of the data, our external function is designed to send a list of four numbers back to *Mathematica*. The first two components are the input interval end-points and the next two components are the end-points of the result. Another reason for designing the communication so that it does not pass back the *Mathematica* `Interval` object but lists of interval end-points is that we want to avoid an extra outward rounding introduced by the *Mathematica* function `Interval` as below. *Mathematica*'s `Take` function is used below to get the last two components of a list.

```
In[4] := Interval[Take[filibRes, -2]] // InputForm
Out[4]//InputForm = Interval[{-0.9899924966004473, 1.0000000000000002}]
```

Since both the machine-precision numbers in *Mathematica* and the arithmetic of `filib++` are based on the same C data type `double` and because the underlying architecture is IEEE-compliant, the communication is transparent without any data conversion. Furthermore, *MathLink* is efficient and will send data in a binary format when communicating with external programs that run on the same computer or when the computers are sufficiently compatible.

Interfacing `filib++` by *Mathematica*, we can overcome the difficulties in using the interval constructors in `filib++` for a correct enclosure of real data. Namely, *Mathematica*'s function `Interval` provides a correct enclosure of intervals involving inexact real data at the end-points.

```
In[5] := Interval[{-1, 2.7}] // InputForm
Out[5]//InputForm = Interval[{-1, 2.7000000000000006}]
```

Then these outwardly rounded end-points are transparently passed to the external program which constructs the `filib++` intervals without rounding.

```
In[6] := filibStdF[Cos, Interval[{-1, 2.7}]] // InputForm
Out[6]//InputForm = {-1., 2.7000000000000006, -0.904072142017063, 1.}
```

While cos() is defined over the whole real line, log() function is not. Let us see what is the result produced by `filib++`.

```
In[7] := filibRes = Take[filibStdF[Log, Interval[{-1, 2.7}]] , -2];
         filibRes // InputForm
Out[8]/InputForm = {-Infinity, 0.9932517730102848}
```

The transparency of floating-point communication between *Mathematica* and `filib++`, explained above, is the reason for obtaining straightforward the *Mathematica* symbol `Infinity` at the corresponding end-point of the result.

Since `Indeterminate` is *Mathematica*'s symbol for the IEEE Not-a-Number and the interval `[NaN, NaN]` is adopted by `filib++` to represent the empty set, the following result is not surprising.

```
In[9] := filibStdF[Log, Interval[{-2, -1}]]
Out[9] = {-2., -1., Indeterminate, Indeterminate}
```

Another example demonstrates again the transparency of the communication and a property of the extended arithmetic of `filib++`. By definition, the point interval `[+infinity]` is represented in `filib++` by the interval `[M, +infinity]`, where `M` is the overflow threshold [10]. Since $\texttt{acoth}(1) = +\infty$, we have

```
In[10] := Take[filibStdF[ArcCoth, Interval[1]], -2]// InputForm
Out[10]//InputForm = {1.7976931348623157*^308, Infinity}
```

Compare this result to the value of *Mathematica*'s global variable `$MaxMachine Number` representing the largest machine-precision number.

```
In[11] := $MaxMachineNumber //InputForm
Out[11]//InputForm = 1.7976931348623157*^308
```

We should mention that passing the symbolic objects $\infty$ and Not-a-Number of the IEEE floating-point formats from *Mathematica* to `filib++` is not straightforward because the communication requires numerical data, while `Infinity` and `Indeterminate` are symbols in *Mathematica*. Therefore, a special design (possibly by catching and handling the exceptions) is required for that purpose.

At the end of using an external program, `Uninstall[link]` should be used to terminate the program represented by the corresponding link object and to remove the *Mathematica* definitions set up by it.

More examples demonstrating *MathLink* connectivity to the library `filib++` can be found in an electronic supplement accessible at

`http://www.math.bas.bg/~epopova/papers/filibMathLink.zip`

The archive contains the source couples `filibStdF.tm/cpp` and `horner.tm/cpp` necessary for building the corresponding *MathLink*-compatible programs under Linux together with a sample `Makefile` and a *Mathematica* notebook demonstrating the execution of these programs from within *Mathematica*. A printable version `filibML.pdf` of the notebook is provided for non-*Mathematica* users.

The *MathLink*-compatible program `horner` is designed to demonstrate the advantage of software interoperability for symbolic preprocessing and checking the

consistency of input data. A `filib++` function `horner()` implements Horner's rule for interval evaluation of a polynomial in one variable when the latter varies within a given interval. This function requires numerical input data: the coefficients of the polynomial and the interval on which the polynomial is evaluated. A communication module is designed and compiled together with the `filib++` function `horner` into a *MathLink* -compatible program allowing the `filib++` function to be called from within a *Mathematica* session. For more details of the implementation see the template file `horner.tm` and the source file `horner.cpp` which we will not comment here since the code is relatively straightforward. For the end-users it is more convenient to introduce a polynomial symbolically. That is why, in the environment of *Mathematica* we have expanded the function `hornerML`, which is named in the template file and maps the external function communicating with the `filib++` function `horner`, by defining a function with the same name `hornerML` but having different type of arguments. The latter function transforms any symbolic-numeric polynomial into a list of its coefficients, does a preliminary check of the user input for consistency, then calls the external function. Thus, *MathLink* connectivity between *Mathematica* and `filib++` allows the latter to exploit all the symbolic-algebraic power of the former. For example, for the *Mathematica* function `hornerML` it does not matter whether or not the polynomial is explicitly given in an expanded form.

```
In[16] := hornerML[(3 - 2x) (x - 1)^2, Interval[{-1., 1.}]]
Out[16] = {-14.000000000000014, 20.000000000000014}
```

It is not only more convenient but also more efficient to check the consistency of the input data within the environment of *Mathematica*. Therefore all argument processing and data-checking is done by the *Mathematica* code, as well as issuing all error messages.

```
In[17] := hornerML[x*y - 2, Interval[{-1, 2}]]
  hornerML::pol: Polynomial of one variable is expected.
Out[17] = hornerML[-2 + x y, Interval[{-1, 2}]]
```

Since the above functionality is achieved entirely by *Mathematica* programming we refer to the electronic supplement for more details and illustrations.

## 5 Integrating C-XSC Programs into *Mathematica*

C-XSC is another open source C++ class library which facilitates the implementation of reliable numerical methods [4], [6], [7]. Beside a lot of predefined numeric data types and the corresponding arithmetic of maximum accuracy for computations in most traditional numerical spaces, the C-XSC environment provides also dotprecision types and several multiple precision data types. A lot of problem-solving numerical routines providing validated results are involved in the distribution of C-XSC (e.g. the former C++ Toolbox for Verified Computing) or provided as external modules or additional software systems [4], [6]. To demonstrate the interoperability between *Mathematica* and C-XSC functions, we have chosen the C-XSC module `parlinsys.cpp` for solving parametric interval linear systems [14]. The same *MathLink* technology can be applied to

other C-XSC functions, e.g. for solving non-parametric (interval) linear systems. The parametric solver was chosen to illustrate the benefit of the *Mathematica* interface for symbolic preprocessing of input data.

The function `ParLinSolve()` from the C-XSC module `parlinsys.cpp` computes guaranteed outer (and inner) inclusions for the exact hull of the united solution set of a parametric linear system involving affine-linear dependencies between interval parameters. Although solving parametric systems, the function requires only numerical input data, namely corresponding sequences of numerical matrices/vectors representing the coefficients for each of the parameters involved, for more details see [14]. We assume that C-XSC module `parlinsys` was successfully compiled and is part of the corresponding include directory of the C-XSC environment. Following *MathLink* technology presented in previous sections, building a *MathLink*- and C-XSC-compatible program, say `ParLinSys`, we first develop a C++ program which includes both libraries and several functions. The details of the implementation code can be found in

`http://www.math.bas.bg/~epopova/papers/cxscMathLink.zip`

Below we present only those aspects of *MathLink* technology that have not been applied and discussed so far.

`ParLinSolveML()` is the function which reads input data from *Mathematica* via variables of fundamental C data types, initializes new variables having the specific C-XSC data types with the incoming data, then calls the C-XSC function `ParLinSolve()` and transforms the computed results into variables of fundamental C data types that will be passed back to *Mathematica*. Since our external function will need to receive expression types that are not among the set handled automatically by `mprep`, we write our own calls (using `MLGet` functions) to get the arguments. For example, by the following code we get an array of floating-point numbers.

```
MLGetDoubleArray(stdlink, &data, &dimensions, &heads, &depth)
 ..... // Allocation C-XSC data types & filling up the parametric matrix
 MLDisownDoubleArray(stdlink, data, dimensions, heads, depth);
```

When an external program gets data from *Mathematica*, it must set up a place to store the data. The first `ML` function above will automatically do this allocation, storing the array in (`double* data`), its dimensions in (`long* dimensions`) and its depth in (`long depth`). After processing these data in the second line above, the memory used to store the array must be released, as in the third code line above. All the input and output data for the external function `ParLinSolveML()` are processed this way. Therefore the `:ArgumentTypes:` and `:ReturnType:` lines in the corresponding template file `ParLinSys.tm` involve the keyword `Manual`. Here is a part of the template file

```
:Function:  ParLinSolveML
:Pattern:   ParLinSolveTB[p_Integer, flag_Integer, ap_?MatrixQ,
                          bp_?MatrixQ, ip_?MatrixQ ]
:Arguments: {p, flag, ap, bp, ip}
:ArgumentTypes:   {Manual}
:ReturnType:      Manual
```

Note, in this case the name of the *Mathematica* calling function `ParLinSolveTB` is different from the name of the external function `ParLinSolveML`. The template file establishes a correspondence between these functions, see Fig. 1.

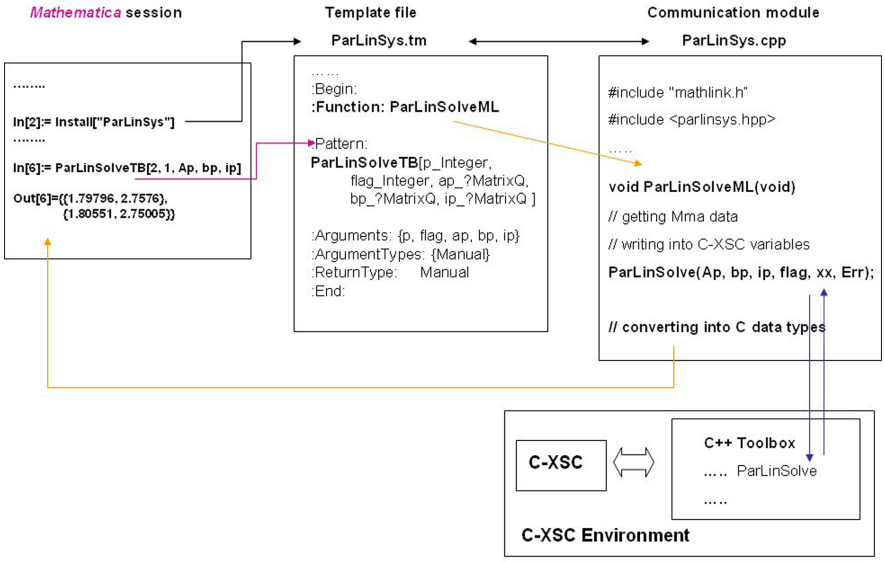


**Fig. 1.** Interaction between *Mathematica* and an external C-XSC program via *MathLink* technology

After developing and compiling the external *MathLink* -compatible program, it can be installed in any *Mathematica* session and the function, defined in the communication module, can be called with appropriate input data.

```
In[1] := lnk = Install["ParLinSys"]
Out[1]= LinkObject["./ParLinSys", 2, 2]
```

Although solving parametric systems, the function requires only numerical input data. For the end-users, it is usually more convenient to define parametric matrices and parametric vectors symbolically as below.

```
In[3] := mat = {{3, p1}, {p1, 3}}; vec = {p2, p2};
```

Therefore, the *Mathematica* interface will be used for symbolically preprocessing the parametric system data. To this end, a new function `parToNumMLData` is defined in the *Mathematica* notebook `cxscML.nb`. This function transforms a parametric matrix, or a parametric vector, whose elements depend affine-linearly on given parameters into a numeric matrix suitable for input for the C-XSC function `ParLinSolve()`, respectively for the external function called by `ParLinSolveTB`. Transforming our symbolic data we get the required numerical input form of the parametric matrix/vector.

```
In[4] := Ap = parToNumMLData[mat, {p1, p2}];
         bp = parToNumMLData[vec, {p1, p2}];
In[6] := pVals = {{1, 2}, {10, 10.5}};
```

For the parameter interval values `pVals`, we just specify a list of interval end-points in the same parameter order `{p1, p2}` as specified by the second argument of `parToNumMLData`. This is because the interval constructors in C-XSC provide directed outward rounding for the interval end-points and the *Mathematica* function `Interval`, if applied, would introduce extra rounding.
Now, we are ready to call our external function.

```
In[7] := ParLinSolveTB[2, 1, Ap, bp, pVals] //InputForm
Out[7] = {{1.792638317329675, 2.762917238225881},
          {1.8018848752730778, 2.753670680282478}}
```

The result of `ParLinSolveTB` is a list involving just the interval end-points, corresponding to the interval vector generated by the C-XSC function. The goal is the same as for the input intervals: to avoid an extra outward rounding introduced by the *Mathematica* function `Interval`.

### Communicating Error Messages

Most of the error-checking for the function `hornerML`, discussed at the end of Section 4, is done by, as well as most of the error messages for this function are issued by, the *Mathematica* code. Some errors, however, can only be detected inside the external functions. Such errors include out-of-memory situations, failed *MathLink* calls, and so on. The external program also can issue some errors that are informative for and should be communicated to the user. *MathLink* can transmit out-of-band data such as exceptions [5], [15]–[17]. Here we demonstrate this extremely important aspect of *MathLink* programming.

Most *MathLink* functions return 0 to indicate an error has occurred, and it is possible to check their return values. If *MathLink* calls are issued after an error has occurred, without clearing the error, the link will probably die. Checking for `MLGet` errors is handled automatically by the code that `mprep` writes for the arguments that are read automatically. This effect can be illustrated by the functions in Section 4.

The code of `ParLinSolveML()` contains calls of `MLGet` functions in order to transmit data types that are not among those handled automatically. If an `MLGet` call fails, the easiest thing to do is simply to abandon the external function call completely and return the symbol `$Failed`. However, it would be more informative to trigger some kind of diagnostic message. The *MathLink* function `MLErrorMessage` returns a string describing the current error and this string is a good candidate for use in an error message to be seen by the user. The following fragment from the communication module `ParLinSys.cpp` detects an error

```
if(!MLGetInteger(stdlink, &p)) PrintMLErrorMessage();
```

and calls the function `PrintMLErrorMessage()` which issues an useful message, then safely bails out of the function call.

```
void PrintMLErrorMessage(void)
{ char err_msg[100];
  sprintf(err_msg, "%s\"%.76s\"%s",
          "Message[ParLinSolveTB::mlink,", MLErrorMessage(stdlink), "]");
  MLClearError(stdlink);   MLNewPacket(stdlink);
  MLEvaluate(stdlink, err_msg);
  MLNextPacket(stdlink);   MLNewPacket(stdlink);
  MLPutSymbol(stdlink, "$Failed");
}
```

Upon detecting the error, the first thing we do in the above function is calling `MLClearError` to attempt to remove the error condition, and then `MLNewPacket` to abandon the rest of the packet containing the original inputs to the function (in case it hasn't been completely read yet). The `sprintf` is used to construct a string of the form:

```
"Message[ParLinSolveTB::mlink, "the text returned by MLErrorMessage"]"
```

which is what is sent to `MLEvaluate`. The message triggered here, `ParLinSolveTB::mlink`, needs to be defined in an `:Evaluate:` line in the template file `ParLinSys.tm` as follows:

```
:Evaluate: ParLinSolveTB::mlink = "Low-level MathLink error: `1`."
```

After the call to `MLEvaluate`, *Mathematica* will send back a `ReturnPacket` containing the return value of the `Message` function (which is the symbol `Null`). To drain this packet off the link we call `MLNextPacket` and then `MLNewPacket` to discard the contents. Since we have several `MLGet` calls in our external code, the collection of the above actions is implemented as a separate function `PrintMLErrorMessage()` transferring the communication error messages.

We demonstrate triggering *MathLink* communication error messages, by calling the function `ParLinSolveTB` with a big value for the first integer argument.

```
In[7] := ParLinSolveTB[2^100, 1, Ap, bp, pVals]
 ParLinSolveTB::mlink: Low-level MathLink error: machine number overflow.
Out[7]= $Failed
```

Any subsequent call of the external function, even with correct data, returns a communication error until the external program is installed again. The same external program can be installed arbitrary many times within a *Mathematica* session. Each installation creates a separate `LinkObject`.

Looking at the code of the external function `ParLinSolveML()`, the extensive check of the input data for consistency should be noticed, e.g.

```
if(dimensions[1] != n) PrintErrorMessage(1);
.............
   if(int_arr[0] > int_arr[1]) PrintErrorMessage(5);
```

Most important is, however, that the C-XSC function `ParLinSolve()` returns also an integer error code. All messages for inconsistent data and computational error messages are passed to *Mathematica* in the same way as the communication

error messages but by another function `PrintErrorMessage()`. The different message strings are defined in separate `:Evaluate:` lines in the template file.

In order to demonstrate the computational error messages in action, we call the function `ParLinSolveTB` with a large interval for the first interval parameter.

```
In[7] := ParLinSolveTB[2, 1, Ap, bp, {{1, 20}, {10, 10.5}}]
  ParLinSolveTB::cond: Verification failed, system is probably ill cond.
Out[7]= $Failed
```

The `ParLinSolveTB::cond` error is triggered whenever the verification iteration is not convergent and the fixed-point parametric iteration fails.

## 6 Conclusion

It is a relatively simple matter to incorporate C-XSC or `filib++` routines into *Mathematica* without any change in the original external code. A communication C/C++ module is necessary to be developed, where *MathLink* functions transmit to and back *Mathematica* expressions via fundamental C data types, the incoming data should initialize new variables of data types specific for the particular external interval environment, and after calling the actual computations to transform the computed results into variables of fundamental C data types that will be passed back to *Mathematica*. *MathLink* protocol allows transparent communication of numerical data without conversion on the same computer. Furthermore, the extended intervals involving $\pm\infty$ at the interval end-points, as well as the empty interval supported in the extended mode of `filib++`, are passed back to *Mathematica* in the same transparent way. By *MathLink* we use the external functions in a way completely integrated into *Mathematica* taking advantage of the good properties of both environments, as demonstrated above. The variety of numerical data types provided in C-XSC, e.g. complex interval arithmetic or dotprecision computations, could be also integrated into *Mathematica* under an appropriate design.

Once provided, *Mathematica* connectivity to external interval software opens up an array on new possibilities for the latter. web*Mathematica* technology, that integrates *Mathematica* into a web server, can be utilized for providing dynamic web access to the external software [12], [13]. The important consequences and benefits from a dynamic web interface concern the development of framework and platforms for distant interval learning and/or remote problem solving.

While the present work demonstrates mainly the interactive and dynamic interfaces for the C-XSC and `filib++` libraries via *MathLink* communication protocol, our further efforts will be directed towards providing interval software interoperability that not only expands the functionality but also improves the performance of the interval tools in solving some practical problems. A current work on communicating symbolic functional expressions between *Mathematica* and C-XSC/`filib++` will allow *interactive* execution of more problem-solving routines that are part of or based on these libraries.

## References


1. Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.): Dagstuhl Seminar 2003. LNCS, vol. 2991. Springer, Heidelberg (2004)
2. Corliss, G.F., Yu, J.: Interval Testing Strategies Applied to COSY's Interval and Taylor Model Arithmetic. In: [1], pp. 91–106
3. Corliss, G.F., Kearfott, R.B., Nedialkov, N., Pryce, J.D., Smith, S.: Interval subroutine library mission. In: Hertling, P., Hoffmann, C.M., Luther, W., Revol, N. (eds.) Reliable Implementation of Real Number Algorithms: Theory and Practice. Dagstuhl Seminar Proceedings, Number 06021, Internationales Begegnungs- und Forschungszentrum für Informatik, Schloss Dagstuhl, Germany (2006)
4. C-XSC library: `http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc_new.html`, solvers: `http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html`
5. Gayley, T.: A MathLink Tutorial. Wolfram Research (2002)
6. Hofschuster, W., Krämer, W., Neher, M.: C-XSC and Closely Related Software Packages. In: Cuyt, A., et al. (eds.) Numerical Validation. LNCS, vol. 5492, pp. 68–102. Springer, Heidelberg (2009)
7. Hofschuster, W., Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. In: [1], pp. 15–35
8. Kreinovich, V.: Interval Computations website, Interval and Related Software, `http://www.cs.utep.edu/interval-comp/intsoft.html`
9. Lerch, M., Tischler, G., Wolff von Gudenberg, J., Hofschuster, W., Krämer, W.: The Interval Library filib++ 2.0 — Design, Features and Sample Programs. Preprint 2001/4, Universität Wuppertal (2001), Library download: `http://www.math.uni-wuppertal.de/org/WRST/software/filib.html`
10. Lerch, M., Tischler, G., Wolff von Gudenberg, J., Hofschuster, W., Krämer, W.: filib++, a Fast Interval Library Supporting Containment Computations. ACM TOMS 32(2), 299–324 (2006)
11. Luther, W., Krämer, W.: Accurate Grid Computing. In: Luther, W., Krämer, W. (eds.) 12th GAMM-IMACS Int. Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2006), Duisburg, September 26-29 (2006)
12. Popova, E.: Web-Accessible Tools for Interval Linear Systems. Proceedings in Applied Mathematics & Mechanics (PAMM) 5(1), 713–714 (2005)
13. Popova, E.: WebComputing Service Framework. Int. Journal Information Theories & Applications 13(3), 246–254 (2006)
14. Popova, E.D., Krämer, W.: Parametric Fixed-Point Iteration Implemented in C-XSC. Preprint BUW-WRSWT 2003/3, Universität Wuppertal (2003), Software download:
    `http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html#plss`
15. Wolfram Research, Inc.: MathLink Reference Guide, Version 2.2., Wolfram Research Inc., Champaign, IL (2003)
16. Wolfram Research, Inc.: MathLink for UNIX Developer Guide, Version 4, Revision 14, Wolfram Research Inc., Champaign, IL, December 15 (2004)
17. Wolfram Research Inc.: Mathematica, Version 5.2, Champaign, IL (2005)