# On the Interoperability between Interval Software

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. In this work we demonstrate *MathLink* technology for integrating C-XSC functions into *Mathematica* and present some of the benefits this approach could bring to both environments.

**Keywords.** Software interoperability, interfacing, interval software, C-XSC, *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 [6]. However there are only a few interval software comparing studies hampered by the diversity in the implementation supporting environments and in the interval data representation. 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 software 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 (API) would facilitate their usage. On another side, most recent interval applications require a combination of diverse 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: [1], directed toward developing of a comprehensive full-featured library of validated routines, and [7], 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 [1]. 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 the general-purpose environments for scientific/technical computing like Matlab, *Mathematica*, Maple, etc. possess several features not attributable to the compiled languages from one side and on another side 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 work we consider the interoperability between *Mathematica* [13] and external C-XSC programs [5], via *MathLink* communication protocol [3]. The focus is on calling external C-XSC functions from within a *Mathematica* session. The goal is to demonstrate some advantages of interval software interoperability. Namely, expanded functionality for both environments, exchanging numerical data without using intermediate files and without any conversion but under dynamics and interactivity in the communication, 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.

## 2   Some Aspects of Interval Software Interoperability

Compiling a library of full-featured, high quality, portable and uniform interval-based tools, as presented in [1], 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. With respect to the development and production process, the approach based on interval software interoperability has the following advantages.

- When building software interoperability the usual tedious and error-prone work of re-implementing an algorithm or a more complex software system is removed providing in the same time a safeguard against the re-implementation bugs.
- In this approach we are usually interested to connect interval software which is already brought to a high quality and efficiency. Therefore, 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 implementation of new algorithms.
- Thus, providing software interoperability requires considerably less development time than building everything from scratch. This approach is especially suitable for re-using complicated methods and large software systems which are already brought to a high quality and efficiency.

Therefore, the overall cost of a software based on interoperability between existing software components is considerably less than a newly compiled software.

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 in some cases considered below.

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 interoperability. 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 environments for scientific/technical computing 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. For many years there is a considerable interaction between symbolic-algebraic and result-verification methods. Embedding of interval data structures, hybrid and result-verification methods in computer algebra systems turn the latter into valuable tool for reliable scientific computing while by applying symbolic-algebraic methods interval computations expand their methodology tools.

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. 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 at 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 [3], [11]–[13]. 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. Some typical uses of *MathLink* are for

- calling functions in an external program from within *Mathematica*,
- calling *Mathematica* from within an external program,
- setting up alternative front ends to *Mathematica*,
- exchanging data between *Mathematica* and external programs,
- exchanging data between concurrent *Mathematica* processes.

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.

If there exists 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. giving an appropriate *MathLink* template for each external function;
2. combine the template with the actual 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 intent is that we 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 obligatory 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. 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 which sets up the external program to be ready to take requests from *Mathematica*. The form of `main` required on different systems may be slightly different, the appropriate form is given in the *MathLink* Developer Kit [13] 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 be linked with the `libML.a` library and the libraries required by our C-XSC application. `mcc` is a script that preprocesses and compiles *MathLink* source files. More information on how to compile and run *MathLink* programs written in C on Unix systems can be found in [12].

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

## 4   Communication Chain C-XSC → *Mathematica*→ Web

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

C-XSC is an open source C++ class library which facilitates the implementation of reliable numerical methods [2], [4], [5]. 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 some dotprecision data 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 [2], [4]. For demonstrating the interoperability between *Mathematica* and C-XSC functions, we have chosen the C-XSC module `parlinsys.cpp` for solving parametric interval linear systems [10]. 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` compute 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 [10]. We assume that C-XSC module `parlinsys` was successfully compiled and is part of the corresponding include directory of the C-XSC environment. Following the *MathLink* technology for building a *MathLink* - and C-XSC-compatible program, let us consider the template file `ParLinSys.tm`.

```
:Evaluate: ParLinSolveTB::usage = "ParLinSolveTB[p, SharpC, Ap, bp, pval]
computes verified enclosure for the solution set of a square parametric
linear system by a C-XSC module."

:Evaluate: ParLinSolveTB::mlink = "Low-level MathLink error: '1'."
:Evaluate: ParLinSolveTB::data = "Incompatible input data."
....................
:Evaluate: ParLinSolveTB::cond = "Verification failed, system is probably
              ill-conditioned."

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

The `:Function:` line specifies the name of the external C routine `ParLinSolveML`. 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. The template file establishes a correspondence between these two functions, see Fig. 1. Note that the arguments in the *Mathematica* function pattern are restricted: the first two to be integers and the next to be matrices. The `:Arguments:` line specifies the expressions to be passed to the external program. In our case these expressions are the same as the variable names on the `:Pattern:` line. 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 *Math-Link* function `MLGetInteger` which transfers C `ints`. 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.

Now, we consider the communication module `ParLinSys.cpp` whose file must be named the same way as the corresponding template file. The program includes both libraries and several functions. The function `ParLinSolveML()` actually communicates the data between *Mathematica* and the C-XSC function `ParLinSolve()`. By the *MathLink* communication protocol the external programs send and receive *Mathematica* expressions using the fundamental C data types. The external programs should not modify the arrays generated by the *MathLink* functions `MLGetRealList()`, `MLGetRealArray()`, etc. Since the external library C-XSC we connect to *Mathematica* use special data types for representing intervals, the main purpose of the communication module is to read the input data from *Mathematica* via variables of fundamental C data types, to initialize new variables having the specific C-XSC data types with the incoming data, then to call the C-XSC function `ParLinSolve()` and then to transform 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 the the set handled automatically by `mprep`, and the function will return different types of results, we write our own calls 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 matrix Ap
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`. Fig. 1 illustrates the interaction between *Mathematica* and the external program via *MathLink* technology.

The source code `ParLinSys.cpp` involves also a main function (standard for the Linux environment we use) and some functions for triggering error messages which will be discussed latter on. More details about the implementation
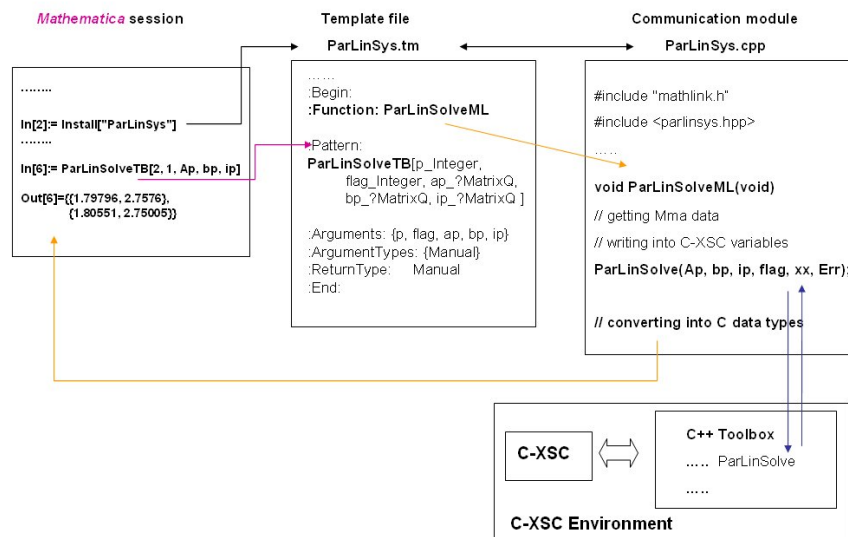
**Fig. 1.** Correspondence between function names and their environments in a *MathLink* connection between *Mathematica* and the C-XSC function ParLinSolve().

can be found in the corresponding source files involved in the electronic supplement cxscMathLink.zip. Since the *MathLink* connection was built in 2003, some source code may seem old fashioned but it is compatible to and works with the latest software versions. The archive involves also a *Mathematica* notebook cxscML.nb demonstrating the execution of the *MathLink*-compatible programs from within a *Mathematica* session. A printable version of this notebook cxscML.pdf is provided for non-*Mathematica* users.

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. The Install function launches the program and opens a link through which the external function can be called from *Mathematica*. 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.

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

Now, the *Mathematica* function, defined in the template file ParLinSys.tm, is ready to be called with appropriate arguments satisfying the corresponding function specification. 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 symbolic preprocessing the parametric system data. A newly developed *Mathematica* function `parToNumMLData` is defined in the *Mathematica* notebook `cxscML.nb`. `parToNumMLData` 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 numerical input form of the parametric matrix/vector required by `ParLinSolveTB`.

```
In[4] := Ap = parToNumMLData[mat, {p1, p2}];
         bp = parToNumMLData[vec, {p1, p2}];
```

For the parameter interval values, 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.

```
In[6] := pVals = {{1, 2}, {10, 10.5}};
```

Now, we are ready to call our external function. *Mathematica* function `InputForm` is used to show all the digits of the result.

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

The result of `ParLinSolveTB` is not a list of *Mathematica* intervals but 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`.

## 4.2   Communicating Error Messages

Most of the argument-checking for the external function can be done by, as well as most of the error messages for this function can be 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. The discussion of our *MathLink*-compatible program so far was missing an extremely important aspect of *MathLink* programming: error-checking, which we present in this section.

*MathLink* is independent of the transport medium and supports a number of different transport mechanisms that have different properties. In addition, *MathLink* can transmit out-of-band data such as exceptions [3], [11]–[13]. 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 any arguments that are read automatically.

For MLGet calls written by ourselves, it is up to us. 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 bail 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 is call 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 simply the symbol Null). We need to drain this packet off the link, so 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.

Now, we demonstrate triggering *MathLink* communication error messages, by calling the function ParLinSolveTB with a very 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

```
In[8] := ParLinSolveTB[2, 1, Ap, bp, pVals]
  ParLinSolveTB::mlink : Low-level MathLink error: MLGet out of sequence.
Out[8]= $Failed
```

until the external program be installed again.

```
In[9] := Install["ParLinSys"]
Out[9]= LinkObject["./ParLinSys", 9, 9]
```

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()`  it can be noticed the extensive check of the input data for consistency, e.g.

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

Most important is, however, that C-XSC function `ParLinSolve()` returns also an integer error code. All messages for inconsistent data and computational error messages are passed to *Mathematica* analogously to the communication error messages but by another function `PrintErrorMessage()`. Of course, the different message strings are defined in separate `:Evaluate:` lines in the template file `ParLinSys.tm`.

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
     conditioned.
Out[7]= $Failed
```

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

### 4.3    Web Interface for *MathLink*-Compatible Programs

Once provided, *Mathematica* connectivity to external interval libraries or problem-solving software opens up an array on new possibilities for the latter. A web*Mathematica* technology, that integrates *Mathematica* into a web server and allows generating of dynamic web content, is utilized for providing dynamic web access to the C-XSC interval linear solvers, see Fig. 2. Since this is not a subject of this work, we refer to [8], [9] for more details. The important consequences and benefits from such a dynamic web interface concern the development of framework and platforms for distant interval learning and/or remote demonstrative problem solving.
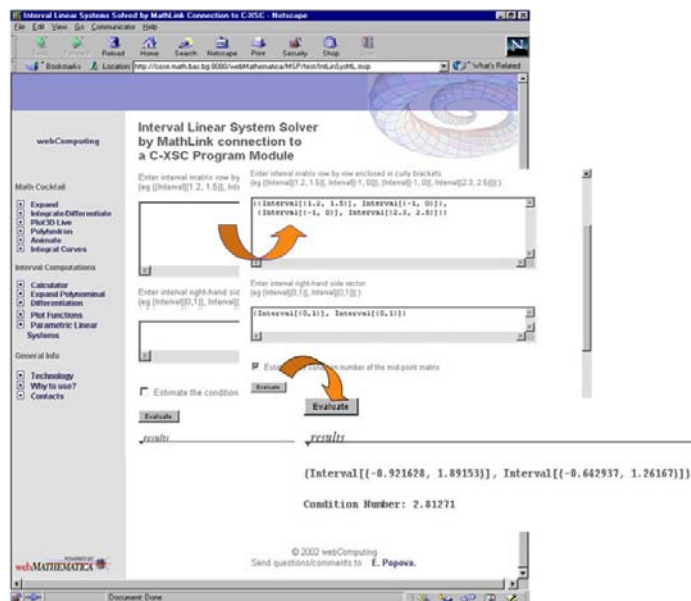
**Fig. 2.** Screen-shot of a dynamic and interactive web interface for the C-XSC interval system solver called from *Mathematica*.

## 5   Conclusion

It is a relatively simple matter to incorporate C-XSC routines into *Mathematica* without any change in the original C-XSC external code. Instead a *MathLink*-compatible C/C++ program should 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 the computed results should be transformed into variables of fundamental C data types that will be passed back to *Mathematica*. *MathLink* protocol allows transparent communication of numerical data without conversion when communicating with external programs that run on the same computer or when the computers are sufficiently compatible. By *MathLink* we use the C-XSC functions in a way completely integrated into *Mathematica* taking advantage of the good properties of both environments, as demonstrated above. We believe that providing interoperability between the existing interval software will bring a desired impact faster than some traditional approaches.

# References

1. 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. Number 06021 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik, Schloss Dagstuhl, Germany (2006) <http://drops.dagstuhl.de/opus/volltexte/2006/712> [date of citation: 2008-02-01].
2. C-XSC library, downloads:
   http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc_new.html,
   solvers: http://www.math.uni-wuppertal.de/~xsc/xsc/cxsc_software.html
3. Gayley, T.: A *MathLink* Tutorial. Wolfram Research, 2002.
   http://library.wolfram.com/infocenter/TechNotes/174/
4. Hofschuster, W.: C-XSC: Highlights and new developments. In: Numerical Validation in Current Hardware Architectures. Number 08021 Dagstuhl Seminar, Internationales Begegnungs- und Forschungszentrum für Informatik, Schloss Dagstuhl, Germany (2008) <http://kathrin.dagstuhl.de/08021/Materials2/>
5. Hofschuster, W., Kraemer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. In Alt, R., Frommer, A., Kearfott, B., Luther, W. (eds.): Numerical Software with Result Verification, Lecture Notes in Computer Science 2991, Springer-Verlag, Heidelberg, pp. 15–35, 2004. Also appeared as Preprint BUW-WRSWT 2003/5, Univ. Wuppertal, 2003.
6. Kreinovich, V.: Interval Computations website, Interval and Related Software.
   http://www.cs.utep.edu/interval-comp/intsoft.html
7. Luther, W., Kramer, W.: Accurate Grid Computing, 12th GAMM-IMACS Int. Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2006), Duisburg, Sept. 26-29, 2006.
8. Popova, E.: Web-Accessible Tools for Interval Linear Systems. Proceedings in Applied Mathematics & Mechanics (PAMM) 5, issue 1, 2005, pp. 713–714.
9. Popova, E.: WebComputing Service Framework. Int. Journal Information Theories & Applications 13(3) 2006, pp. 246–254.
10. 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
11. Wolfram Research, Inc.: MathLink Reference Guide, Version 2.2., Wolfram Research Inc., Champaign, IL, 2003.
12. Wolfram Research, Inc.: MathLink for UNIX Developer Guide, Version 4, Revision 14, Wolfram Research Inc., Champaign, IL, December 15, 2004.
13. Wolfram Research Inc.: *Mathematica*, Version 5.2, WRI, Champaign, IL, 2005.