Communicating Functional Expressions from *Mathematica* to C-XSC*

Evgenija D. Popova[1] and Walter Krämer[2]

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

[2] WRSWT, Bergische Universität Wuppertal, Faculty of Mathematics and Natural Sciences, Gaußstr. 20, D-42097 Wuppertal, Germany
kraemer@math.uni-wuppertal.de

Abstract. This work focuses on a mechanism (and software) which communicates (via *MathLink* protocol) and provides compatibility between the representation of nonlinear functions specified as *Mathematica* expressions and objects of suitable classes supported by the C-XSC automatic differentiation modules. The application of the developed communication software is demonstrated by *MathLink* compatible programs embedding in *Mathematica* the C-XSC modules for automatic differentiation as packages. The design methodology, some implementation issues and the use of the developed software are discussed.

1 Introduction

Modeling, design and simulation of real-life problems often require integration of symbolic and numerical computations, visualization tools, etc. Providing interoperability between the general-purpose environments for scientific/technical computing (like *Mathematica*, Maple, etc.), which possess several features not attributable to the compiled languages, and the interval software, developed in some compiled language for efficiency reasons, is highly desirable due to the many positive consequences for both environments [10]. Computer-assisted proofs is another field that requires a collaboration of both environments [4].

Although many interval methods are brought to reliable, high-quality and fast implementations, they remain isolated software systems. The software comparing studies are also hampered by the diversity in the implementation supporting environments and in the interval data representation (see e.g. [3] and the references given therein). Communication based on file reading/writing operations to exchange ordinary text is not suitable for interval computations. The problems that have to be solved are enlisted in [3]. Due to the necessity of avoiding decimal-to-binary/binary-to-decimal input/output conversions of floating-point data, the most suitable approach when providing interoperability between interval software is that based on communication protocols [10].

* This work is partially supported by the DFG grant GZ: KR1612/7-1, AOBJ: 570029.

K. Fukuda et al. (Eds.): ICMS 2010, LNCS 6327, pp. 354–365, 2010.

In order to promote the interoperability between interval supporting environments as an alternative to building complicated systems from scratch we have initiated a concrete study. Its aim is establishing a general framework for delivering external specific arithmetic tools and implementations of interval algorithms, provided by the C++ class library C-XSC [8,9], in the general purpose environment of *Mathematica* [12] via the communication protocol *MathLink* [5]. *Mathematica* was chosen as a genera-purpose computer algebra system providing the most flexible support of interval arithmetic, e.g. supporting exact interval arithmetic. *Mathematica* also has its own high-level interface standard for interprogram communication. It is desirable to expand *Mathematica*'s interval functionality and to help new developments by connecting to external interval libraries. C-XSC is an open source library facilitating the implementation of reliable numerical methods. It supplies some arithmetic tools that are not available in other interval libraries, such as accurate dot product expressions, or complex interval arithmetic that is not supported in *Mathematica*. A lot of problem solvers delivering validated results are involved[1] in the distribution of C-XSC or provided as external software.

A preliminary research demonstrates the transparency in the communication of floating-point (interval) data between *Mathematica* and C-XSC [10]. The article contains a lot of technical details giving the reader a better feeling how easy or complicated it is to establish a connection between the two environments. However, some important problems such as isolating zeros of nonlinear functions (one- and multi-dimensional), guaranteed global optimization (one- and multi-dimensional), and automatic differentiation (AD) of interval functions, for which there are C-XSC modules, as well as many other problems, are formulated in terms of nonlinear functions. Therefore, the communication of functional expressions from *Mathematica* to C-XSC is an important step toward creating a general interface that allows using all application functions delivered with C-XSC from within *Mathematica*.

The goal of this paper is to discuss some issues related to communication of functional expressions from *Mathematica* to C-XSC and to present the software tools developed for this purpose. Since all of C-XSC problem solvers for nonlinear functions are built on top of the AD arithmetic supported by three C-XSC modules, the particular goal of the present work is to provide compatibility between the functional expressions specified in *Mathematica* and the specific data type objects used in the C-XSC AD modules. Section 2 presents shortly the C-XSC AD modules. Some aspects of C-XSC that are important for the present development are mentioned. The notion of expression in *Mathematica* is presented in the next section. Section 4 is devoted to the basic software communicating nonlinear functions, specified in *Mathematica*, to the respective C-XSC class objects. The design principles and the implementation are discussed. As a proof of concept illustrating the application of this tool, three interfacing *MathLink* programs are developed. They embed in *Mathematica* the corresponding C-XSC modules for automatic differentiation as packages. Section 5 describes their use.

[1] The former C++ Toolbox for Verified Computing [7].

2 C-XSC Background

C-XSC [8,9] contains modules for AD based on interval operations to get guaranteed enclosures for the function value and the derivatives up to order two in the one dimensional case, resp., the gradient/hessian in the n-dimensional case.

The C-XSC module `ddf_ari` implements AD arithmetic for functions $f : \mathbb{R} \to \mathbb{R}$ which are twice continuously differentiable. This module supports the class `DerivType` including operations and elementary functions for the differentiation arithmetic up to second order. An object of type `DerivType` is implemented as a triple of intervals representing the function value and the values of first and second derivatives, respectively. To get enclosures for the true values of the function and its derivatives, the differentiation arithmetic is based on interval arithmetic. That is, the three components of the `DerivType` object are replaced by the corresponding interval values, and all arithmetic operations and function evaluations are executed in interval arithmetic.

Any application program using the C-XSC module `ddf_ari` must contain a C++ function, with argument type `DerivType` and result type `DerivType`, which specifies the functional expression. For example, a function $f(x) = x(4+x)/(3-x)$ should be specified in a C++ function as follows.

```
DerivType f(const DerivType& x)
{ return x*(4.0 + x)/(3.0 - x); }
```

Let `x, fx` be declared as `DerivType` variables and `y(123.0)` be an interval variable. The following two objects of class DerivType contain a variable with value y and the representation of $f(x)$ in the point y, respectively.

```
x = DerivVar(y);        fx = f(x);
```

Then, `fValue(fx), dfValue(fx), ddfValue(fx)` get the function and the derivative values in the point interval $y = 123.0$. The C-XSC functions `fEval(), dfEval(), ddfEval()` simplify the mechanism of function and derivative evaluation. Thus, `ddfEval(f, y, fy, dfy, ddfy)` evaluates the three enclosures.

The module `hess_ari` supports classes `HessType` and `HTvector`, as well as operators and elementary functions for interval differentiation arithmetic of gradients and Hessians of scalar-valued functions $f : \mathbb{R}^n \to \mathbb{R}$. The module can also be used to compute Jacobians of vector-valued functions $f : \mathbb{R}^n \to \mathbb{R}^n$. The module `grad_ari` supports classes `GradType` and `GTvector`. These classes avoid the storage overhead for the Hessian components. The design, implementation and the application of the C-XSC AD modules are similar, see [7]. Table 1 summarizes the supported data types and the evaluation routines.

C-XSC AD modules include derivative constants and variables with both real and interval arguments, predefined arithmetic operations $\circ \in \{+, -, \times, /\}$ for any combination of real, interval and `DerivType` (or `HessType`, `GradType`, resp.), and the elementary functions enlisted in Table 2 that are predefined for the corresponding `DerivType`, `HessType` or `GradType` argument. Although the present version of C-XSC contains a lot of other mathematical functions, these

Table 1. C-XSC modules for AD: supported data types and predefined routines

C-XSC module	`ddf_ari`	`grad_ari`	`hess_ari`
scalar func.	$\mathbb{R} \to \mathbb{R}$	$\mathbb{R}^n \to \mathbb{R}$	$\mathbb{R}^n \to \mathbb{R}$
object type	`DerivType`	`GradType`	`HessType`
arg. type	`DerivType`	`GTvector`	`HTvector`
predefined functions	`fValue(DerivType)`	`fValue(GradType)`	`fValue(HessType)`
	`dfValue(DerivType)`	`gradValue(GradType)`	`gradValue(HessType)`
	`ddfValue(DerivType)`		`hessValue(HessType)`
	`fEval()`	`fEvalG()`	`fEvalH()`
	`dfEval()`	`fgEvalG()`	`fgEvalH()`
	`ddfEval()`		`fghEvalH()`
vector func.		$\mathbb{R}^n \to \mathbb{R}^n$	$\mathbb{R}^n \to \mathbb{R}^n$
object type		`GTvector`	`HTvector`
arg. type		`GTvector`	`HTvector`
predefined functions		`fValue(GTvector)`	`fValue(HTvector)`
		`JacValue(GTvector)`	`JacValue(HTvector)`
		`fEvalJ()`	`fEvalJ()`
		`fJEvalJ()`	`fJEvalJ()`

functions cannot be used explicitly in the functional expressions specifying function objects of types `DerivType`, `HessType` or `GradType`.

Some rules for getting true enclosures in connection with conversion (decimal to binary) errors (see [7, Chapters 2.5, 3.7], [11]) should also be applied when specifying functional expressions used by the C-XSC AD modules.

Table 2. Elementary functions supported by C-XSC AD arithmetic. Power function implements integer power.

exp	ln	sin	cos	tan	cot	asin	acos	atan	acot
sqr	power	sqrt	sinh	cosh	tanh	coth	asinh	acosh	atanh	acoth

3 Expressions in *Mathematica*

At the core of *Mathematica* is the foundational idea that everything — data, programs, formulas, graphics, documents — can be represented as symbolic expressions. Although they often look very different, *Mathematica* represents all of these things in one uniform way. They are all expressions. A prototypical example of a *Mathematica* expression is `f[x,y,...]`. It is not necessary to write expressions in this form. For example, `x+y` is also an expression. When it is typed in `x+y`, *Mathematica* converts it to the standard form `Plus[x,y]`. Then, when it prints it out again, it gives it as `x+y`. All expressions – whatever they may represent – ultimately have a uniform structure. The function `FullForm` gives the full functional form of an expression, without shortened syntax.

```
In[1] := FullForm[1 + x^2 + x/y]
Out[1]//FullForm = Plus[1, Power[x, 2], Times[x,Power[y,-1]]]
```

Whenever *Mathematica* knows that a function is associative, it tries to remove parentheses (or nested invocations of the function) to get the function into a standard "flattened" form. The standard form is the one in which all the terms are in a definite order, corresponding roughly to alphabetical order [12]. A function like addition or multiplication is not only associative, but also commutative, which means that expressions like $a+c+b$ and $a+b+c$ with terms in different orders are equal. One reason for putting expressions into standard forms is that if two expressions are really in standard form, it is obvious whether or not they are equal. However, it is not always desirable that expressions be reduced to the same standard form. This is especially valid in interval arithmetic, where the machine addition and multiplication are not associative, and in range computation, where different forms of an expression can give different quality of the range enclosure, cf. [6, Chapter 3.3]. The *Mathematica* function `Hold[expr]` provides "wrappers" inside which the expressions remain unevaluated.

```
In[3] := x*Hold[b*a] //FullForm
Out[3]//FullForm = x Hold[b a]
```

When the user types in an expression, *Mathematica* automatically applies its large repertoire of rules for transforming expressions. The function `Hold` can also be used to prevent an expression from being simplified.

4 Communication of Functional Expressions

In this section we present a *MathLink* compatible basic C++ software, called `ADExpressions`, which communicates dynamically (via the *MathLink* protocol) functional expressions defined in *Mathematica* and in evaluation converts them into objects of C-XSC AD classes representing the function (and its derivatives).

4.1 Algorithm

The main goal of `ADExpressions` is to provide compatibility between the *Mathematica* representation of a functional expression and the representation used by C-XSC AD modules. Therefore, the software should read a *Mathematica* expression via appropriate *MathLink* tools, parse the expression providing compatibility between the operations and the elementary functions in both environments, and represent the parsed expression appropriately so that the evaluation generates objects of suitable C-XSC AD classes (`DerivType, GradType, GTVector, HessType, HTVector`). This work is organized in two separate main stages.

1. *Initialization.* The input expression coming from *Mathematica* via the *MathLink* protocol is parsed and stored in an appropriate C++ representation.
2. *Evaluation.* Read from *Mathematica* an interval value for each of the expression variables, substitute the expression variables (in the internal expression

representation) with objects of respective classes `DerivType, GradType, GTVector, HessType, HTVector`, where the objects are instantiated with the input values, and evaluate the expression according to its internal representation and by the respective automatic differentiation arithmetic. The result of the evaluation is an object of the appropriate C-XSC class.

The two stages are separated in order to provide re-usability of the evaluation step. In the initialization stage the expression formula is parsed and converted from the canonical *Mathematica* form to a postfix (reverse Polish) notation [1]. The postfix notation is chosen to allow a faster evaluation of the expression than the infix notation. For example, an expression having the canonical *Mathematica* form `Plus[Sin[y], x]` is stored as $\{y, Sin, x, Plus\}$, where the number of arguments for each function is known.

A nonlinear function represented as a *Mathematica* expression may contain

- numerical constants: integer, real, interval, the *Mathematica* constants `E`, `Pi`; The interval constants should be represented by the *Mathematica* object `Interval[{a, b}]`, where the interval end-points `a, b` can be integer or real numbers. Exact singletons cannot be used at the interval end-points but can be used in the other parts of a functional expression. For simplicity *Mathematica* numbers with heads `Rational` and `Complex`[2], as well as another syntax of the interval object are not admissible for the functional expression.
- variables having interval values; The names of the variables should follow the *Mathematica* convention for names, or be *Mathematica* indexed variables with the syntax `name[i]`.
- arithmetic operations, a set of elementary functions, and the *Mathematica* function `Hold` enlisted in Table 3.

To provide compatibility between the *Mathematica* expression and its C-XSC representation, the following transformations are performed during the parsing.

T.1 All the admissible functions in an input *Mathematica* expression have one or two arguments, except for the functions `Plus` and `Times` which are associative for exact arguments. During the parsing multi-argument functions `Plus` and `Times` are replaced by multiple two-argument functions. Therefore, we assume that all functions in the expression have one or two arguments.

T.2 All non-interval numerical constants are stored as C-XSC point intervals. Corresponding C-XSC interval constants containing the true value of the *Mathematica* constants `E` and `Pi` are used.

T.3 When all arguments of a function are constants, it is evaluated by the corresponding C-XSC function and the enclosing interval constant is stored in the internal representation.

T.4 For every two-argument function there are three internal representations corresponding to all combinations of the argument types (constant or expression). The commutative functions have two internal representations. Thus, the function `Plus`, which is commutative, is represented internally

[2] C-XSC AD modules do not support complex arithmetic.

Table 3. *Mathematica* operator symbols and functions, recognized by the expression parser, and their equivalent C-XSC representation

Mathematica operator or function	C-XSC representation	Comments
$+, -, \times, /$	$+, -, \times, /$	Sect. 3, T.1
`Hold[Subtract[x,y]]`	`x-y`	
`Hold[Divide[x,y]]`	`x/y`	
`^, Power[x, y_Integer]`	`sqr(x)`	if y=2
	`power(x, y)`	other integer `y`
`Power[E, y], Hold[Exp[x]]`	`exp(x)`	
`^, Power[x, y]`	`exp(ln(x)*y)`	otherwise
`Hold[Sqrt[x]]`	`sqrt(x)`	
`Log[x], Log[E, x]`	`ln(x)`	nat. logarithm
`Log[b,x]`	`ln(x)/ln(b)`	log to base `b`
`Sin[x],Cos[x],Tan[x],Cot[x]`	$\sin(x), \cos(x), \tan(x), \cot(x)$	
`Sinh[x],Cosh[x],Tanh[x],Coth[x]`	$\sinh(x), \cosh(x), \tanh(x), \coth(x)$	
`ArcSin[x], ArcCos[x], ArcTan[x]`	$\mathrm{asin}(x)$, $\mathrm{acos}(x)$, $\mathrm{atan}(x)$	
`ArcCot[x],ArcSinh[x],ArcCosh[x]`	$\mathrm{acot}(x)$, $\mathrm{asinh}(x)$, $\mathrm{acosh}(x)$	
`ArcTanh[x], ArcCoth[x]`	$\mathrm{atanh}(x)$, $\mathrm{acoth}(x)$	
`Csc[x], Sec[x]`	`1/sin(x), 1/cos(x)`	
`Hold[expr]`	expr	`Hold` dropped

by two classes, say `plus(expr1, expr2)` and `plus(const, expr)`. During the initialization, depending on the type of the particular arguments, it is determined which of the two classes will be used in the internal representation. In case of a constant argument, the second representation class `plus(const, expr)` is used and it is initialized with the particular constant value.

T.5 *Mathematica* functions like logarithm to a base, power function with real exponent, secant, cosecant, which are not supported by the C-XSC AD modules, are transformed to relevant C-XSC expressions, see Table 3.

The evaluation stage consists of three actions:

- Reading a numerical interval for each variable. It is assumed that an interval comes from *Mathematica* as a two-dimensional list of the interval end-points. The order $\leq$ of the interval end-points is not checked by the communication software, so a C-XSC runtime error can arise if the relation is not fulfilled.
- The variables involved in the functional expression are replaced by a variable of appropriate C-XSC class which contains the concrete interval value(s). Which C-XSC class will be used depends on the function under evaluation and the particular C-XSC automatic differentiation module, see Table 1.
- The evaluation of an expression in postfix notation is done by using a stack. The postfix expression to be evaluated is scanned from left to right. Variables or constants are pushed onto the stack. When an operator/function is encountered, the indicated action is performed (by the particular C-XSC

differentiation arithmetic) using the top elements of the stack, and the result replaces the operands on the stack.

4.2 Implementation

All functions and classes of the basic communication software `ADExpressions` are defined in the namespace `mlcxsc`[3]. Hence, it is required to qualify each identifier of the module with `mlcxsc::` or to use the directive `using namespace mlcxsc`. `ADExpressions` consists of two basic template classes:

- `MLCXSCFunctionScalar<class T>` which communicates, represents and evaluates one- or n-variate scalar functions $f(x) : \mathbb{R} \longrightarrow \mathbb{R}$ or $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}$;
- `MLCXSCFunctionVector<class T>` which communicates, represents and evaluates vector-valued functions $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}^n$.

The template parameter `T` in the class `MLCXSCFunctionScalar` can be one of the three C-XSC data types `DerivType`, `GradType`, or `HessType`, corresponding to the respective C-XSC AD modules. Vector-valued functions are treated as n functions $f_i : \mathbb{R}^n \longrightarrow \mathbb{R}$. The two template classes are intended to work with the communication protocol *MathLink* since in their initialization it is supposed that the functional expression is in the *MathLink* queue.

The two basic template classes have four public methods (member functions):

- `Init()` initializes an object, called representation list, with the functional expression which should be evaluated. In case of some error it throws an exception with specified corresponding error message.
- `operator()` evaluates the function computing its range and the ranges of its first and second derivatives for the given interval values of the variable(s). In case of some error it throws an exception.
- `IsInitialized()` returns `true` if the representation list is initialized and `false` otherwise. The representation list is initialized whenever the function `Init()` has finished successfully, then we can call `operator()`.
- `Invalidate()` makes a representation list not valid.

Copy constructor and `operator=` for the template classes `MLCXSCFunction Scalar` and `MLCXSCFunctionVector` are not implemented.

In order to facilitate catching and communicating error messages, an exception class `MLCXSCFunctionException` is defined as follows.

```
class MLCXSCFunctionException: public std::logic_error
{public:
 enum ErrorCode
 {ecIllegalArgumentError,// Init,operator called with illegal arguments
  ecNotInitializedError,// calling operator() before executing Init()
  ecExpressionError,    // error in parsing a Mma expression by Init()
  ecInternalError       // internal error that should not normally occur
```

[3] "`ml`" in the abbreviation `mlcxsc` comes from *MathLink*.

```
 };
 private:
   ErrorCode mErrorCode;
 public:
   MLCXSCFunctionException(ErrorCode aErrorCode,
             const string& aMessage):std::logic_error(aMessage)
   {      mErrorCode = aErrorCode;                }
   ErrorCode GetErrorCode() const {  return mErrorCode; }
};
```

The error messages are defined in sufficient details at the place they arise and are associated with particular error code. The error code has self-explanatory enumerated values commented in the above source code. The four values try to minimize the number of exception types that will be triggered to *Mathematica* and correspond to four main stages of the algorithm. Different error messages, regarding the same stage of the algorithm, are communicated by one error code. For example, all kind of errors that arise during the expression parsing are associated with and communicated by the `ecExpressionError` code. Some of these messages are: `Unknown variable`, `Unknown function`, `Unknown Data Type`, wrong syntax of an interval in the expression, etc.

The software `ADExpressions` is designed as an auxiliary tool. Although based on *MathLink* protocol for communicating the expressions, this basic software does not contain functions installable in *Mathematica*. Its purpose is to facilitate the development of such functions, that is, the implementation of external *MathLink* programs integrating C-XSC routines for, or routines based on, AD arithmetic. `ADExpressions` can be downloaded from

http://www.math.bas.bg/~epopova/software/ADExpressions.zip

For the sake of readability the distribution is split into three files: `types.h`, `expression.h`, `expression.cpp`. Everyone who wants to use the software shall include `expression.h` in the source file of the own *MathLink* program. Examples of installable *MathLink* programs which use `ADExpressions` are contained in the archive `ADpackages.zip` which can be downloaded from the above site. The archive contains three external *MathLink* compatible programs `ddfAri, gradAri, hessAri` embedding in *Mathematica* the corresponding C-XSC AD modules by using the `ADExpressions` software. Any of the three *MathLink* programs is an excellent example of how to use the basic communication software `ADExpressions` and provides a framework for further developments.

5 Embedding C-XSC AD Modules in *Mathematica*

MathLink [5,12] and the implemented basic software communicating functional expressions allow any external C-XSC function to be called from within *Mathematica*. Any of the *MathLink* compatible programs `ddfAri, gradAri, hessAri` presents an interesting example of extending the *Mathematica* kernel. These programs involve an interaction between the external C-XSC code, the parsing

software communicating functional expressions, the *Mathematica* package code, and the kernel. Due to the lack of space some interesting design and *MathLink* programming issues that arose in their development are discussed in [11].

Once the external *MathLink* programs are compiled, the executable files (containing the corresponding package) can be launched in any *Mathematica* session.

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

Table 4 enlists the *Mathematica* functions providing interface for the respective C-XSC functions from Table 1. All three packages can be installed in the same *Mathematica* session and be used simultaneously.

Table 4. *Mathematica* packages, resp. functions, interfacing the C-XSC modules for automatic differentiation `ddf_ari, hess_ari, grad_ari`

	MathLink compatible packages	
`ddfAri`	`gradAri`	`hessAri`
`SetFunction[var, func]`	`SetFScalarGrad[{vars}, func]`	`SetFScalarHess[{vars}, func]`
`ReadyFunctionQ[ ]`	`ReadyScalarGradQ[ ]`	`ReadyScalarHessQ[ ]`
`fValue[int]`	`fValueScalarG[{ints}]`	`fValueScalarH[{ints}]`
`dfValue[int]`	`gradValueG[{ints}]`	`gradValueH[{ints}]`
`ddfValue[int]`		`hessValue[{ints}]`
`dfEval[var, fun, int]`	`fgEvalG[{vars}, func, {ints}]`	`fgEvalH[{vars}, func, {ints}]`
`ddfEval[var, fun, int]`		`fghEvalH[{vars}, func, {ints}]`
	`SetFVectorGrad[{vars}, {funcs}]`	`SetFVectorHess[{vars}, {funcs}]`
	`ReadyVectorGradQ[ ]`	`ReadyVectorHessQ[ ]`
	`fValueVectorG[{ints}]`	`fValueVectorH[{ints}]`
	`JacValueG[{ints}]`	`JacValueH[{ints}]`
	`fJEvalJGrad[{vars},{funcs},{ints}]`	`fJEvalJHess[{vars},{funcs},{ints}]`

Now, we initialize a scalar function $f = 2 - \sum_{i=1}^{10} x_i^2$ in 10 variables and evaluate it in the interval $[-1, 1]$ for each variable. Since the variables are many, we use indexed variables and *Mathematica* functions for generating the functional expression and the lists of variable names and variable values.

```
In[5]:= SetFScalarGrad[Table[x[i],{i,10}], 2-Sum[x[i]^2, {i,10}]]
```

The output `Null` of the successful evaluation of `SetFScalarGrad` is not visible.

```
In[6] := ReadyScalarGradQ[ ]
Out[6] = True
In[7] := fValueScalarG[Table[{-1, 1}, {i, 10}]]
Out[7] = {-8., 2.}
```

For the evaluation of vector-valued functions and their derivatives, C-XSC modules for AD impose the restriction (which is followed by the interface) that the number of specified variables must be equal to the dimension of the function.

```
In[10] := SetFVectorGrad[{x}, {x, 1 - x, x^2}]
MLCXSCErrorGrad::illargs : Illegal arguments: Init: The number
of variables must be equal to the  dimension of the function.
Out[10] = $Failed
```

However, one variable name can be repeated many times in the list of variable names, see [11]. Dummy variables can be also used which makes possible the guaranteed evaluation of constants.

```
In[13] := fJEvalJGrad[{x}, {Hold[Sqrt[3]]}, {{1,2}}]
Out[13] = {{{1.73205, 1.73205}}, {{{0., 0.}}}}
```

The implementation of C-XSC AD modules uses the standard error handling (runtime error) of the interval library employed in case of some error during the function or the derivatives evaluation, see [7]. C-XSC runtime errors during the evaluations cannot be checked and cause closing the connection but the corresponding C-XSC message is displayed in a separate `stderr` window.

The *Mathematica* interface for C-XSC routines provides more possibilities for controlling the conversion input errors. Inexact quantities can be enclosed either by the *Mathematica* function `Interval`, or by an interval enclosure in C-XSC (using the techniques discussed in [7]) and wrapping in the `Hold` function.

```
In[16]:= SetFScalarGrad[{x}, Hold[Divide[23,Interval[{10,10}]]]*x]
        fValueScalarG[{{1, 2}}] // FullForm
Out[17]//FullForm = List[2.3‘,4.6000000000000005‘]
```

The representation of a mathematical function in different expression forms is very important for the interval computations. The symbolic interface of C-XSC enables evaluation of functions in different forms via the `Hold` function. The next simple example demonstrates the lack of inverse interval addition.

```
In[18] := fgEvalG[{x}, Hold[x-x], {{-1,1}}]
Out[18] = {{-2., 2.}, {{0., 0.}}}
```

6 Conclusion

The integration of C-XSC automatic differentiation modules in the environment of *Mathematica* via *MathLink* communication protocol brings dynamics and interactivity in the execution of C-XSC modules working with nonlinear functions. This saves the compilation time for each concrete function in the analysis of practical problems. The paper shows that it might be much easier and cheaper to connect a general-purpose computer algebra (CA) system to a sophisticated interval library than to implement interval algorithms from scratch using the programming language of the corresponding CA system (even if some interval tools are already provided by the CA system). `ADExpressions` can facilitate the implementation of any other *MathLink* program providing new numerical methods based on AD. By the interoperability approach we obtain an

integrated environment which combines the interactivity, symbolic and visualization tools of *Mathematica* with the rigorousness and the speed of C-XSC interval functionality.

While the syntactical interoperability between *Mathematica* and C-XSC is provided essentially by *MathLink* communication protocol, the major efforts should be devoted to providing semantic interoperability — the ability to automatically interpret the information exchanged, as defined by the end users, meaningfully and accurately in order to produce useful results. AD of complex-valued functions is not supported in C-XSC and therefore by `ADExpressions`. However, the present version of C-XSC contains a lot of other mathematical functions, besides those enlisted in Table 2, which can be evaluated for complex-valued real/interval variables. A separate work could provide a more generic framework allowing communication of real or complex-valued interval functions evaluated rigorously by the expanded set of C-XSC elementary functions. Other interesting directions of future investigation are the ability of both systems to exchange dot-product expressions and results, and the embedding of C-XSC complex interval arithmetic/solvers not available in *Mathematica*.

Although the current work focuses on one-way communication — the embedding of C-XSC functionality in the environment of *Mathematica*, it can help the respective work on using the *Mathematica* kernel from within C-XSC programs.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools, 2nd edn. Pearson/Addison Wesley (2007)
2. Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.): Numerical Software with Result Verification. LNCS, vol. 2991. Springer, Heidelberg (2004)
3. Corliss, G.F., Yu, J.: Interval Testing Strategies Applied to COSY's Interval and Taylor Model Arithmetic. In: [2], pp. 91–106
4. Frommer, A.: Proving conjectures by use of interval arithmetic. In: Kulisch, U., et al. (eds.) Perspectives on Enclosure Methods, pp. 1–13. Springer, Wien (2001)
5. Gayley, T.: A MathLink Tutorial. Wolfram Research, Champaign (2002)
6. Hansen, E.: Global Optimization Using Interval Analysis. Marcel Dekker, New York (1992)
7. Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: C++ Toolbox for Verified Computing: Basic Numerical Problems. Springer, Heidelberg (1995)
8. Hofschuster, W., Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. In: [2], pp. 15–35
9. Klatte, R., Kulisch, U., Lawo, C., Rauch, M., Wiethoff, A.: C-XSC, A C++ Class Library for Extended Scientific Computing. Springer, Heidelberg (1993)
10. Popova, E.: *Mathematica* Connectivity to Interval Libraries filib++ and C-XSC. In: Cuyt, A., Krämer, W., Luther, W., Markstein, P. (eds.) Numerical Validation in Current Hardware Architectures. LNCS, vol. 5492, pp. 117–132. Springer, Heidelberg (2009)
11. Popova, E., Krämer, W., Russev, M.: Integration of C-XSC Automatic Differentiation in Mathematica, Preprint 3/2010, IMI-BAS, Sofia (March 2010), `http://www.math.bas.bg/~epopova/papers/10-preprintAD.pdf`
12. Wolfram Research Inc.: Mathematica, Version 5.2, Champaign, IL (2005)